Developer Guide

Amazon Timestream

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon Timestream: Developer Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

..... xv

Amazon Timestream for LiveAnalytics 1

- Timestream for LiveAnalytics key benefits 1
- Timestream for LiveAnalytics use cases 2
- Getting started with Timestream for LiveAnalytics 2
- Amazon Timestream for LiveAnalytics availability change 3
  - Amazon Timestream for LiveAnalytics availability change 3
  - Migration Guide 5
- How it works 22
  - Concepts 22
  - Architecture 24
  - Writes 29
  - Storage 43
  - Queries 45
  - Scheduled queries 49
  - Timestream Compute Unit (TCU) 49
- Accessing Timestream for LiveAnalytics 58
  - Sign up for an AWS account 59
  - Provide Timestream for LiveAnalytics access 59
  - Using the console 60
  - Using the AWS CLI 65
  - Using the API 69
  - Using the AWS SDKs 72
- Getting started 77
  - Tutorial 77
  - Sample application 79
- Code samples 80
  - Write SDK client 81
  - Query SDK client 84
  - Create database 86
  - Describe database 90
  - Update a database 94
  - Delete database 98
  - List databases 102

Create table ... 107
Describe table ... 115
Update table ... 119
Delete table ... 123
List tables ... 127
Write data ... 132
Run query ... 187
Run UNLOAD query ... 212
Cancel query ... 235
Create batch load task ... 238
Describe batch load task ... 252
List batch load tasks ... 257
Resume batch load task ... 262
Create scheduled query ... 266
List scheduled query ... 282
Describe scheduled query ... 286
Execute scheduled query ... 289
Update scheduled query ... 293
Delete scheduled query ... 296
Using batch load ... 299
Concepts ... 299
Prerequisites ... 300
Best practices ... 302
Preparing a batch load data file ... 303
Data model mappings ... 304
Using batch load with the console ... 308
Using batch load with the CLI ... 312
Using batch load with the SDKs ... 320
Using batch load error reports ... 320
Using scheduled queries ... 321
Benefits ... 322
Use cases ... 322
Example ... 323
Concepts ... 323
Schedule expressions ... 327
Data model mappings ... 331

Notification messages ... 350
Error reports ... 356
Patterns and examples ... 360
Using UNLOAD ... 459
Benefits ... 460
Use cases ... 460
Concepts ... 461
Prerequisites ... 471
Best practices ... 473
Example use case ... 475
Limits ... 479
Using query insights ... 480
Benefits ... 480
Optimizing data access ... 481
Enabling query insights in Amazon Timestream ... 486
Optimizing queries ... 487
Working with AWS Backup ... 491
How it works ... 492
Creating backups ... 496
Restoring backups ... 498
Copying backups ... 499
Deleting backups ... 500
Quotas and limits ... 500
Customer-defined partition keys ... 501
Using customer-defined partition keys ... 501
Getting started with customer-defined partition keys ... 502
Checking partitioning schema configuration ... 506
Updating partitioning schema configuration ... 512
Advantages of customer-defined partition keys ... 515
Limitations of customer-defined partition keys ... 515
Customer-defined partition keys and low cardinality dimensions ... 515
Creating partition keys for existing tables ... 516
Timestream for LiveAnalytics schema validation with custom composite partition keys ... 516
Tagging resources ... 519
Tagging restrictions ... 519
Tagging operations ... 520

Security ... 521
Data protection ... 522
Identity and access management ... 525
Logging and monitoring ... 563
Resilience ... 567
Infrastructure security ... 567
Configuration and vulnerability analysis ... 568
Incident response ... 568
VPC endpoints ... 568
Security best practices ... 572
Working with other services ... 574
Amazon DynamoDB ... 575
AWS Lambda ... 575
AWS IoT Core ... 577
Amazon Managed Service for Apache Flink ... 581
Amazon Kinesis ... 583
Amazon MQ ... 590
Amazon MSK ... 591
Amazon Quick ... 594
Amazon SageMaker AI ... 598
Amazon SQS ... 600
DBeaver ... 601
Grafana ... 606
SquaredUp ... 607
Open source Telegraf ... 608
JDBC ... 613
ODBC ... 629
VPC endpoints ... 637
Best practices ... 637
Data modeling ... 638
Security ... 655
Configuring Timestream for LiveAnalytics ... 656
Writes ... 657
Queries ... 658
Scheduled queries ... 659
Client applications and supported integrations ... 660

General ... 661
Metering and cost optimization ... 661
Writes ... 661
Storage ... 664
Queries ... 665
Cost optimization ... 665
Monitoring with Amazon CloudWatch ... 666
Troubleshooting ... 682
Handling WriteRecords throttles ... 682
Handling rejected records ... 682
Troubleshooting UNLOAD ... 683
Timestream for LiveAnalytics specific error codes ... 685
Quotas ... 687
Default quotas ... 687
Service limits ... 688
Supported data types ... 692
Batch load ... 692
Naming constraints ... 693
Reserved keywords ... 695
System identifiers ... 697
UNLOAD ... 697
Query language reference ... 698
Supported data types ... 699
Built-in time series functionality ... 702
SQL support ... 716
Logical operators ... 725
Comparison operators ... 727
Comparison functions ... 727
Conditional expressions ... 730
Conversion functions ... 732
Mathematical operators ... 733
Mathematical functions ... 733
String operators ... 736
String functions ... 737
Array operators ... 740
Array functions ... 741

Bitwise functions ... 749
Regular expression functions ... 751
Date / time operators ... 756
Date / time functions ... 758
Aggregate functions ... 775
Window functions ... 790
Sample queries ... 794
API reference ... 807
Document history ... 807
Amazon Timestream for InfluxDB ... 819
DB instances ... 819
DB instance classes ... 821
DB instance class types ... 821
Hardware specifications ... 821
Instance Storage ... 823
InfluxDB storage types ... 823
Instance sizing ... 823
Regions and Availability Zones ... 824
Regions availability ... 826
Regions design ... 828
Availability Zones ... 828
Billing ... 829
Setting up ... 829
Sign up for AWS ... 829
Setting up ... 830
Determine requirements ... 832
VPC access ... 834
Getting started ... 835
Creating and connecting to a Timestream for InfluxDB instance ... 836
Creating a new operator token for your InfluxDB instance ... 850
Migrating data from self-managed InfluxDB to Timestream for InfluxDB ... 850
Preparation ... 851
How to use scripts ... 852
Migration Overview ... 855
Configuring a DB instance ... 859
Creating a DB instance ... 859

Settings for DB instances ... 862
Connecting to an Amazon Timestream for InfluxDB DB instance ... 866
Working with read replica clusters ... 901
Instance class availability ... 901
Read replica cluster architecture ... 902
Parameter groups ... 903
Replica lag ... 904
Availability and durability ... 905
Read replicas cluster overview ... 905
Creating a read replica cluster ... 910
Connecting to a read replica DB cluster ... 923
Modifying a read replica cluster ... 925
Rebooting a read replica cluster ... 929
Creating CloudWatch alarms to monitor Timestream for InfluxDB ... 930
Read replica licensing through AWS Marketplace ... 934
Managing DB instances ... 939
Updating DB instances ... 939
Maintenance windows ... 941
Deleting a DB instance ... 944
Rebooting a DB instance ... 946
Multi-AZ DB instance deployments ... 947
Setup to view InfluxDB Logs on Timestream Influxdb Instances ... 951
Monitoring and Configuration Optimization ... 953
Backup and restore ... 988
Overview ... 988
Key concepts ... 989
Creating a DbBackup ... 989
Configuring automated backups ... 990
Listing and viewing backups ... 993
Restoring from a DbBackup ... 994
Deleting backups ... 997
Performance impact ... 997
Relationship to service-managed backups ... 997
Security and encryption ... 998
Limitations ... 998
Pricing ... 999

Tagging resources ... 999
Tagging restrictions ... 999
Best practices for Timestream for InfluxDB ... 1000
Optimize writes to InfluxDB ... 1000
Design for performance ... 1001
Troubleshooting ... 1004
Warning of "dev" version not recognized ... 1004
Migration failed during restoration stage ... 1004
Amazon Timestream for InfluxDB basic operational guidelines ... 1004
DB instance RAM recommendations ... 1005
Amazon EventBridge events ... 1005
Overview ... 1005
Event categories ... 1006
Source types ... 1007
Event schema ... 1007
Setting up event notifications ... 1009
Common patterns ... 1011
Instance event reference ... 1012
Cluster event reference ... 1014
Troubleshooting ... 1017
Limitations ... 1017
Pricing ... 1018
Security ... 1018
Overview ... 1019
Database authentication with Amazon Timestream for InfluxDB ... 1022
How Timestream for InfluxDB uses secrets ... 1024
Data protection ... 1031
Identity and Access Management ... 1038
Logging and monitoring ... 1077
Compliance validation ... 1080
Resilience ... 1080
Infrastructure security ... 1081
Configuration and vulnerability analysis in Timestream for InfluxDB ... 1081
Incident response ... 1082
Amazon Timestream for InfluxDB API and interface VPC endpoints (AWS PrivateLink) ... 1082
Security best practices ... 1085

Working with other services 1087
DBeaver 1087
Grafana 1088
API reference 1093
Document history 1093
Amazon Timestream for InfluxDB 3 1105
What is Timestream for InfluxDB 3? 1105
DB Clusters 1105
Cost Optimization Strategies 1113
AWS Regions and Availability Zones 1113
Getting started with InfluxDB 3 1113
Setting up Timestream for InfluxDB 3 1113
Creating and connecting to a Timestream for InfluxDB instance 1115
Configuring an InfluxDB 3 cluster 1117
Creating a DB cluster in Timestream 1117
S3 VPC Endpoint for Private Clusters 1118
Settings for DB clusters 1120
Parameter Groups for DB Clusters in Amazon Timestream 1121
Core and Enterprise versions 1151
Deployment models 1154
Endpoints and connectivity 1155
Manage InfluxDB 3 Clusters 1160
Create a cluster 1161
Update a cluster 1167
Upgrade Core to Enterprise 1172
Delete a cluster 1175
Reboot a cluster 1178
Scaling a cluster 1179
Maintenance windows 1182
How maintenance windows work 1182
Management types 1182
Preferred maintenance window 1183
Timezone 1184
During maintenance 1185
API response fields 1185
CLI examples 1186

Considerations ... 1188
Backup and restore ... 1188
Overview ... 1188
Key concepts ... 1189
Creating a DbBackup ... 1189
Configuring automated backups ... 1190
Listing and viewing backups ... 1192
Restoring from a DbBackup ... 1193
Deleting backups ... 1195
Performance impact ... 1196
Relationship to service-managed backups ... 1196
Security and encryption ... 1196
Limitations ... 1197
Pricing ... 1197
Amazon EventBridge events ... 1197
Writing data to your InfluxDB 3 cluster ... 1198
Line protocol overview ... 1199
API endpoints overview ... 1200
Client libraries and integrations ... 1203
Telegraf integration ... 1205
Best practices for writing data ... 1209
Querying data from InfluxDB 3 ... 1210
Query methods overview ... 1211
Authentication options for the v3 HTTP Query API ... 1213
Response formats for the v3 HTTP Query API ... 1213
SQL Query Examples ... 1214
Query performance best practices ... 1216
Client library query support ... 1216
Compare SQL and InfluxQL query capabilities ... 1217
Query optimization features ... 1218
InfluxDB schema design recommendations ... 1223
InfluxDB data structure ... 1224
Schema design best practices ... 1225
Processing engine plugins ... 1226
InfluxData certified plugins ... 1226
Plugin types and trigger specifications ... 1227

Create triggers 1227
Configure triggers 1228
Manage triggers 1230
Distributed deployment considerations 1231
Best practices 1231
Monitor plugin execution 1232
Available plugins 1232
Common configuration patterns 1241
Best practices for plugins 1243
Monitor plugin execution 1243
Troubleshoot common issues 1244
Custom plugins 1244
How custom plugins work 1245
Prerequisites 1245
Step 1: Create a parameter group with the plugin repository 1246
(Optional) Create the repository access secret 1247
Step 2: Apply the parameter group to your cluster 1249
Step 3: Create a trigger that runs your custom plugin 1249
Use multiple plugin repositories 1250
Update custom plugins 1251
Security considerations 1252
Python dependencies 1255
Limitations 1255
Monitor custom plugin execution 1255
Troubleshoot custom plugins 1256
Customer managed keys 1257
Overview 1257
Supported key types 1257
Creating a cluster with a customer managed key 1258
Setting up the KMS key policy 1258
Viewing the encryption key 1260
Monitoring key usage with CloudTrail 1260
Key revocation and deletion 1261
Migrating existing clusters to use a customer managed key 1261
Limitations 1261
Pricing 1262

InfluxDB 3 FAQ 1262
General 1263
Clusters and instances 1266
Writing and querying data 1269
Configuration and parameter groups 1272
Schema design 1274
Processing engine 1274
Custom plugins 1275
Storage and billing 1277
Security and access 1279

For similar capabilities to Amazon Timestream for LiveAnalytics, consider Amazon Timestream for InfluxDB. It offers simplified data ingestion and single-digit millisecond query response times for real-time analytics. Learn more here.

What is Amazon Timestream for LiveAnalytics?

Amazon Timestream for LiveAnalytics is a fast, scalable, fully managed, purpose-built time series database that makes it easy to store and analyze trillions of time series data points per day. Timestream for LiveAnalytics saves you time and cost in managing the lifecycle of time series data by keeping recent data in memory and moving historical data to a cost optimized storage tier based upon user defined policies. Timestream for LiveAnalytics's purpose-built query engine lets you access and analyze recent and historical data together, without having to specify its location. Amazon Timestream for LiveAnalytics has built-in time series analytics functions, helping you identify trends and patterns in your data in near real-time. Timestream for LiveAnalytics is serverless and automatically scales up or down to adjust capacity and performance. Because you don't need to manage the underlying infrastructure, you can focus on optimizing and building your applications.

Timestream for LiveAnalytics also integrates with commonly used services for data collection, visualization, and machine learning. You can send data to Amazon Timestream for LiveAnalytics using AWS IoT Core, Amazon Kinesis, Amazon MSK, and open source Telegraf. You can visualize data using Quick, Grafana, and business intelligence tools through JDBC. You can also use Amazon SageMaker AI with Timestream for LiveAnalytics for machine learning.

Timestream for LiveAnalytics key benefits

The key benefits of Amazon Timestream for LiveAnalytics are:

- *Serverless with auto-scaling* - With Amazon Timestream for LiveAnalytics, there are no servers to manage and no capacity to provision. As the needs of your application change, Timestream for LiveAnalytics automatically scales to adjust capacity.
- *Data lifecycle management* - Amazon Timestream for LiveAnalytics simplifies the complex process of data lifecycle management. It offers storage tiering, with a memory store for recent data and a magnetic store for historical data. Amazon Timestream automates the transfer of data from the memory store to the magnetic store based upon user configurable policies.
- *Simplified data access* - With Amazon Timestream for LiveAnalytics, you no longer need to use disparate tools to access recent and historical data. Amazon Timestream for LiveAnalytics's purpose-built query engine transparently accesses and combines data across storage tiers without you having to specify the data location.

- *Purpose-built for time series* - You can quickly analyze time series data using SQL, with built-in time series functions for smoothing, approximation, and interpolation. Timestream for LiveAnalytics also supports advanced aggregates, window functions, and complex data types such as arrays and rows.
- *Always encrypted* - Amazon Timestream for LiveAnalytics ensures that your time series data is always encrypted, whether at rest or in transit. Amazon Timestream for LiveAnalytics also enables you to specify an AWS KMS customer managed key (CMK) for encrypting data in the magnetic store.
- *High availability* - Amazon Timestream ensures high availability of your write and read requests by automatically replicating data and allocating resources across at least 3 different Availability Zones within a single AWS Region. For more information, see the Timestream Service Level Agreement.
- *Durability* - Amazon Timestream ensures durability of your data by automatically replicating your memory and magnetic store data across different Availability Zones within a single AWS Region. All of your data is written to disk before acknowledging your write request as complete.

Timestream for LiveAnalytics use cases

Examples of a growing list of use cases for Timestream for LiveAnalytics include:

- Monitoring metrics to improve the performance and availability of your applications.
- Storage and analysis of industrial telemetry to streamline equipment management and maintenance.
- Tracking user interaction with an application over time.
- Storage and analysis of IoT sensor data.

Getting started with Timestream for LiveAnalytics

We recommend that you begin by reading the following sections:

- **Tutorial** - To create a database populated with sample data sets and run sample queries.
- **Amazon Timestream for LiveAnalytics concepts** - To learn essential Timestream for LiveAnalytics concepts.
- **Accessing Timestream for LiveAnalytics** - To learn how to access Timestream for LiveAnalytics using the console, AWS CLI, or API.

- **Quotas** - To learn about quotas on the number of Timestream for LiveAnalytics components that you can provision.

To learn how to quickly begin developing applications for Timestream for LiveAnalytics, see the following:

- Using the AWS SDKs
- Query language reference

Amazon Timestream for LiveAnalytics availability change

After careful consideration, we have made the decision to close new customer access to Amazon Timestream for LiveAnalytics, effective 6/20/25. This change will not impact customer workloads running with Amazon Timestream for LiveAnalytics. Existing customers with an active payer account currently using the service may continue to add new users and linked accounts under that payer account. AWS continues to invest in security, availability, and performance improvements for Amazon Timestream for LiveAnalytics. We recommend that new customers evaluate Amazon Timestream for InfluxDB as an alternative due to its similar functionality.

Topics

- Amazon Timestream for LiveAnalytics availability change
- Migration Guide

Amazon Timestream for LiveAnalytics availability change

Since time-series applications have unique requirements and characteristics, we offer a broad framework to help you evaluate various alternatives before diving into specific implementation details. This high-level guidance serves as a foundation for your decision-making process, with more detailed steps and practical implementations to be covered in subsequent sections.

Alternative services evaluation

Large-scale analytics, high cardinality, or new workloads (recommended)

We recommend Amazon Timestream for InfluxDB 3 Enterprise for most LiveAnalytics migrations. InfluxDB 3 is built on Apache Arrow, DataFusion, and Parquet with Amazon S3 object storage, supporting unlimited cardinality, standard SQL and InfluxQL queries, multi-node

clusters (up to 15 nodes), and a built-in Python processing engine. It is the recommended target for workloads with more than 1 million series, complex analytical queries, or any new greenfield deployment.

Low-latency operational monitoring (less than 1 million series)

We recommend Amazon Timestream for InfluxDB 2, if your Timestream for LiveAnalytics table has less than 10 million cardinality (series keys), meaning the unique combinations of Amazon Timestream for LiveAnalytics concepts or if you can reduce your table's cardinality under 10 million. Amazon Timestream for InfluxDB 2 provides single-instance or Multi-AZ deployments with single-digit millisecond query response times, Flux and InfluxQL query support, and tasks (equivalent to Scheduled queries).

Prefer using SQL instead of InfluxQL

Amazon Timestream for InfluxDB 3 supports native SQL (standard SQL via Apache DataFusion) as well as InfluxQL. If your LiveAnalytics workloads use SQL, Amazon Timestream for InfluxDB 3 is the direct migration path—there is no need to move to any other SQL based engine. InfluxDB 3's SQL implementation is purpose-built for time-series data with optimized time-based functions, GROUP BY time intervals, and columnar storage that outperforms general-purpose relational databases for time-series analytics.

Require high-scale data ingestion (exceeding 1 million records per second)

While Amazon Timestream for InfluxDB 3 Enterprise is built for large-scale time-series workloads, a single cluster cannot currently support more than 1 million records per second. For workloads exceeding this threshold, data would need to be sharded across multiple InfluxDB 3 clusters. If sharding is not an option, NoSQL engines like Amazon DynamoDB could be a good alternative for use cases that involve low complexity analytics.

Before beginning your data migration to the chosen alternate AWS service, it is crucial to assess several key factors that will significantly influence your migration strategy and its ultimate success. These evaluations will help shape your approach, identify potential challenges, and ensure a smoother transition during the migration process.

Data selection and retention considerations

Assess your data migration scope by defining exact retention requirements. Consider whether you need to migrate the complete historical dataset, recent data only (such as the last 30, 60, or 90 days), or specific time-series data segments. This decision should be guided by three key

factors: regulatory compliance requirements, analytical needs of your business, and practical considerations around migration complexity and costs.

Query pattern compatibility analysis

While Amazon Timestream for InfluxDB 3 supports SQL, the SQL dialect does not match LiveAnalytics 1:1. Thorough testing of all business-critical queries is required before migration. We recommend running your existing query set against InfluxDB 3 in a dev cluster to identify syntax differences and validate performance meets your requirements.

Data transformation planning

Before migrating, pay close attention to schema mapping to ensure proper data alignment and structural consistency between source and target systems, and accurate data type conversions specifically tailored for time-series data. These components work together to ensure data quality, optimize performance, and maintain functionality across different system architectures. In addition, consider any specialized indexing patterns and system-specific optimizations to guarantee efficient data access and retrieval.

Continuity and downtime management

Since data migration inherently causes operational disruption, developing a comprehensive switchover strategy is crucial for success. Few best practices to consider in the migration plan to minimize downtime are:

- Implement temporary parallel processing systems where possible to maintain business continuity.
- Schedule migrations during low-traffic periods such as weekends or overnight hours.
- Establish well-tested rollback procedures for quick recovery in case of unexpected issues.

Migration Guide

This guide presents approaches for migrating time-series data from Amazon Timestream for LiveAnalytics to Amazon Timestream for InfluxDB 3 (recommended for most workloads), Amazon Timestream for InfluxDB 2 (for low-latency operational workloads), and to Aurora or RDS PostgreSQL (for non-time-series use cases) with an intermediate layer for Amazon S3.

In this guide, we walk through following steps:

1. Export your data from Timestream for LiveAnalytics to Amazon S3.

2. Migrating to Amazon Timestream for InfluxDB 3 (recommended for most workloads).
3. Ingesting data to Amazon Timestream for InfluxDB 2.
4. Ingesting data to PostgreSQL.

Topics

- Exporting Timestream data to Amazon S3
- Timestream for InfluxDB as a Target
- Aurora/RDS Postgres as a target

Exporting Timestream data to Amazon S3

Irrespective of the target service for migration, we recommend following the below best practices for exporting your Timestream for LiveAnalytics data to Amazon S3, creating a durable intermediate storage layer that serves as the foundation for subsequent database-specific ingestion.

To reliably export data from Timestream for LiveAnalytics tables to Amazon S3, we recommend using Timestream for LiveAnalytics export tool, which uses the Timestream UNLOAD feature — designed for large-scale data exports.

Timestream for LiveAnalytics export tool

Time-based chunking strategy

Time-based chunking is essential when migrating large volumes of time-series data. This approach breaks down the export process into manageable units that can be independently processed and re-tried on failures, significantly reducing migration risks. It creates checkpoints for easier progress tracking and adds the ability to resume after interruptions. For organizations with continuous data ingestion, this allows newer data to be exported in separate time chunks, enabling better coordination between ongoing operations and migration. The tool uses day-based chunking, storing each day's data with S3 bucket prefix for efficient management. Additionally, chunking can be based on hour, day, month, or year.

Monitoring migration

The tool provides an option to capture the migration statistics in a DynamoDB table, tracking metrics such as configurations used, records exported, and other data points for validating the completeness of your migration. We recommend monitoring these metrics closely during your

migration and validation. You can also use the logging provided within your orchestration script, capturing execution timestamps, chunk boundaries, and any error conditions encountered. The tool also provides SNS notification if you want to integrate your downstream system to take action on failures.

Recommendations and best practices

The Timestream for LiveAnalytics export tool provides a flexible and robust solution for exporting data to S3 with various configuration options tailored to your target system requirements. If your target is Timestream for InfluxDB, use Parquet format without compression to ensure compatibility with ingestion scripts. For optimal tracking and monitoring, enable DynamoDB logging and configure SNS notifications to receive alerts about export failures or completions.

The tool leverages the Timestream for LiveAnalytics UNLOAD feature while overcoming its partition for query limitations by automatically exporting data in batches based on your specified time range. You can customize data partitioning by hour, day, month, or year, day being the default. Each partition must remain under approximately 350 GB to avoid memory-related errors, such as query computation exceeding maximum available memory. For example, if your yearly data exceeds 350 GB, consider using monthly partitions or even more granular options like daily or hourly partitioning. If you choose hourly and still get a "The query computation exceeds maximum available memory" error, you can reduce the number of partitions, making sure your exports are successful.

The tool offers flexibility in the scope of export, allowing you to export a single table, an entire database, or all databases in your account. For more specialized requirements, such as exporting multiple specific databases, you can build a custom wrapper around this automation. Additionally, you can choose to export the most recent data first by enabling the reverse chronological order option. When restarting after failures, you can either continue with the same migration tag to keep all files under the same S3 prefix or omit the tag to create files under a new prefix. As the tool exports the data in batches, if you encounter failures we recommend starting from the failed batch rather than restarting from the original start time. If you don't specify an end timestamp, the tool automatically uses the current timestamp (UTC) to ensure consistent exports and validation.

Basic commands

Example: Export a table with DynamoDB logging enabled

```
python3.9 unload.py \
    --export-table \
```

```
    --database Demo \
    --table Demo \
    --start-time '2020-03-26 17:24:38' \
    --enable-dynamodb_logger true
```

Example: Export entire database

```
python3.9 unload.py \
    --export-database \
    --database Demo \
    --start-time '2020-03-26 17:24:38'
```

Example: Export all databases

```
python3.9 unload.py \
    --export-all_databases \
    --start-time '2020-03-26 17:24:38'
```

Example: Advanced export with more options

```
python unload.py \
    --export-table \
    --database MyDB \
    --table MyTable \
    --start-time '2024-05-14 00:00:00' \
    --end-time '2025-05-14 00:00:00' \
    --partition month \
    --export-format PARQUET \
    --compression GZIP \
    --region us-east-1 \
    --s3-uri s3://my-bucket \
    --enable-dynamodb_logger \
    --sns-topic_arn arn:aws:sns:region:account-id:topic-name
```

For more information, see the unload script's README.

Timestream for InfluxDB as a Target

Amazon Timestream for InfluxDB is a managed time-series database service on AWS that uses open-source InfluxDB APIs for real-time applications. It offers easy setup, operation, and scaling, delivering queries with single-digit millisecond response times.

The first step for determining whether Timestream for InfluxDB is an appropriate migration target for your use-case is determining the cardinality of your Timestream for LiveAnalytics table. We have developed a script that calculates table cardinality in Timestream for LiveAnalytics. This calculation serves two purposes:

1. Checks if the cardinality is under 10 million, which will help determine whether Timestream for InfluxDB can handle your use-case.
2. Helps you decide which Timestream for InfluxDB Instance type to use.

Cardinality in InfluxDB is the number of unique measurements, tags, and field key combinations in an InfluxDB bucket. Refer to Timestream for InfluxDB's documentation on cardinality management to understand how exceeding recommended limits can degrade query performance and increase memory consumption. Benchmark your anticipated query patterns against representative data samples before finalizing your instance selection to ensure your queries remain performant post-migration. Pay attention to memory-intensive aggregation queries that might behave differently than in Timestream for LiveAnalytics. When migrating from Timestream for LiveAnalytics, carefully select your InfluxDB instance specifications based on your dataset's cardinality as this directly impacts performance and resource requirement. We recommend considering other destinations if your data cardinality is more than 10 million.

Cardinality calculation script overview

The cardinality calculation script calculates the cardinality of a Timestream for LiveAnalytics table. If the cardinality is under 10 million, the script recommends a Timestream for InfluxDB instance type. Using the default schema mapping, cardinality is calculated by computing the total unique combinations of dimensions and measure name. Choosing the right line protocol tags (equivalent to dimensions in Timestream for LiveAnalytics) helps you automatically index your data and filter your data efficiently using tags. The script also provides the option to exclude specific dimensions when calculating cardinality. If applicable to your case that is, if you are not using certain dimensions for filtering data in SQL queries (specifically not using them as predicates) then you can exclude these dimensions from the cardinality calculation. Later, you can ingest them as fields (equivalent to measures in Timestream for LiveAnalytics) in the next steps of migration.

Prerequisites and installation

See the Prerequisites section and installation in the cardinality script's README.

Basic usage

To determine the cardinality of a table, example_table, in the database example_database the script can be used in the following way:

Example

```
python3 cardinality.py \
    --table-name example_table \
    --database-name example_database
```

This produces the following output:

```
Cardinality of "example_database"."example_table": 160
Your recommended Timestream for InfluxDB type is: db.influx.medium
```

Recommendations

The script automatically scans the entire table to calculate cardinality while offering time filter options for optimal query execution. We suggest implementing time filters when your data involves consistent dimensions and when analyzing distinct dimension variations across the entire table yields similar results to analyzing specific time ranges. This approach ensures efficient and performant query execution.

For more information, see the cardinality script's README.

Ingesting data from Amazon S3 to Timestream for InfluxDB automation

After the Timestream for LiveAnalytics export tool completes the unload process, the next step in the automation process begins. This automation uses InfluxDB's import tools to transfer the data into its specialized time-series structure. The process transforms Timestream's data model to match InfluxDB' s concepts of measurements, tags, and fields. Finally, it loads the data efficiently using InfluxDB's line protocol.

The workflow for completing a migration is separated into four stages:

1. Unload data using Timestream for LiveAnalytics export tool.
2. Data transformation: Converting Timestream for LiveAnalytics data into InfluxDB line protocol format (Based on the schema defined after the cardinality assessment) using Amazon Athena.
3. Data ingestion: Ingest the line protocol dataset to your Timestream for InfluxDB instance.
4. Validation: Optionally, you can validate that every line protocol point has been ingested (Requires `--add-validation-field true` during data transformation step).

Data Transformation

For data transformation, we developed a script to convert Timestream for LiveAnalytics exported data parquet format into InfluxDB's Line Protocol format using Amazon Athena. Amazon Athena provides a serverless query service and a cost-effective way to transform large volumes of time-series data without requiring dedicated compute resources.

The script does the following:

- Loads exported Timestream for LiveAnalytics data from an Amazon S3 bucket into an Amazon Athena table.
- Performs data mapping and transformation from the data stored in the Athena table into line protocol and stores it in the S3 bucket.

Data Mapping

The following table shows how Timestream for LiveAnalytics data is mapped to line protocol data.

Timestream for LiveAnalytics Concept	Line Protocol Concept
Table Name	Measurement
Dimensions	Tags
Measure name	Tag (Optional)
Measures	Fields
Time	Timestamp

Prerequisites and Installation

See the Prerequisites and Installation sections in the transformation script's README.

Usage

To transform data stored in the bucket example_s3_bucket from the Timestream for LiveAnalytics table example_table in example_database, run the following command:

```
python3 transform.py \
```

```
        --database-name example_database \
        --tables example_table \
        --s3-bucket-path example_s3_bucket \
        --add-validation-field false
```

After the script is completed,

- In Athena, the table example_database_example_table will be created, containing Timestream for LiveAnalytics data.
- In Athena, the table lp_example_database_example_table will be created, containing Timestream for LiveAnalytics data transformed to line protocol points.
- In the S3 bucket example_s3_bucket, within the path `example_database/example_table/unload-<%Y-%m-%d-%H:%M:%S>/line-protocol-output`, line protocol data will be stored.

Recommendations

Refer to the transformation script's README for more details on the latest usage of the script and outputs are required for later steps of the migration, such as validation. If you excluded dimensions in order to improve cardinality, adjust the schema to reduce cardinality by using the `--dimensions-to-fields` argument to change particular dimensions to fields.

Adding a Field for Validation

For information on how to add a field for validation, see the Adding a Field for Validation section in the transformation script's README.

Migrating to Amazon Timestream for InfluxDB 3

Amazon Timestream for InfluxDB 3 Enterprise is the recommended migration target for most LiveAnalytics workloads. It provides standard SQL and InfluxQL query support, unlimited cardinality, multi-node scaling, and a built-in Python processing engine.

Key differences from LiveAnalytics

Feature	LiveAnalytics	InfluxDB 3 Enterprise
Architecture	Serverless, fully managed	Instance-based clusters (db.influx.* family)

Feature	LiveAnalytics	InfluxDB 3 Enterprise
Storage	Automatic memory + magnetic tiers	Amazon S3 object storage (Apache Parquet)
Query language	Custom Timestream SQL dialect	Standard SQL + InfluxQL
Write protocol	AWS SDK (WriteRecords API)	Line Protocol over HTTPS (port 443)
Scaling	Automatic	Manual: choose instance size + node count
Cardinality	Automatic partitioning	Unlimited (no series key index)
Processing	Scheduled queries	Built-in Python processing engine (plugins, triggers)
Auth model	IAM for data-plane	Token-based authentication (Admin Token at provisioning)
Pricing	Per-write, per-query, per-GB stored	Per-instance-hour + included Amazon S3 I/O + InfluxData license (Marketplace)
Availability SLA	99.99%	99.9% (multi-node, multi-AZ)
Regions	10 regions (frozen)	19+ regions (expanding)

Instance types available

Instance	vCPUs	Memory
db.influx.large	2	8 GB
db.influx.xlarge	4	16 GB
db.influx.2xlarge	8	32 GB

Instance	vCPUs	Memory
db.influx.4xlarge	16	64 GB
db.influx.8xlarge	32	128 GB
db.influx.12xlarge	48	192 GB

Editions

InfluxDB 3 Core

Single-node, open-source engine, no license fee. Good for development, testing, and smaller workloads.

InfluxDB 3 Enterprise

Multi-node clusters (up to 15 nodes), InfluxData Marketplace license required. Recommended for production workloads at scale. Supports configurable node modes (INGEST, QUERY, COMPACT, PROCESS, or ALL).

Feature mapping

LiveAnalytics feature	InfluxDB 3 equivalent
Scheduled Queries (downsampling)	Processing Engine scheduled triggers (Python plugins)
Memory Store (recent data)	WAL buffer + Last Value Cache
Magnetic Store (historical data)	Amazon S3 Parquet files (compacted)
Multi-measure records	Multiple fields per line protocol point
Dimension-based partitioning	Tag-based automatic partitioning
UNLOAD to Amazon S3	Export API (table-level Parquet export)
Cross-region replication	Not available (use multi-AZ within region)
AWS Backup integration	Automated backups (snapshot-based)

LiveAnalytics feature	InfluxDB 3 equivalent
VPC Endpoints	Private clusters deployed in customer VPC
AWS KMS encryption	Encryption at rest (AWS managed keys; CMK support available)

Steps to migrate from LiveAnalytics to InfluxDB 3

1. **Assess your workload:** Identify write throughput (records/sec), query patterns (point queries vs. aggregations), cardinality, and data retention needs. If using Scheduled Queries in LiveAnalytics, plan to replace with InfluxDB 3 Processing Engine plugins.
2. **Provision your InfluxDB 3 cluster:** Choose Core (single-node) or Enterprise (multi-node) based on throughput needs. Create via AWS Console, AWS CLI, or CloudFormation. Select instance size based on write throughput and query concurrency.
3. **Update write path:** LiveAnalytics uses the `WriteRecords` API (AWS SDK). InfluxDB 3 uses **Line Protocol** over HTTPS. Update Telegraf outputs, application code, or IoT Rules to write via Line Protocol to the cluster endpoint. Batch writes with 5,000–10,000 points per batch, sorted by tag keys alphabetically.
4. **Update query path:** LiveAnalytics custom SQL maps to InfluxDB 3 standard SQL with key differences: use `time` column directly (no `measure_value::double` pattern), use `WHERE time > now() - INTERVAL '1 hour'` instead of `ago()`. Use Flight SQL (port 443) for programmatic access or InfluxQL for Grafana dashboards.
5. **Migrate historical data:** Export from LiveAnalytics using `UNLOAD` to Amazon S3 (Parquet or CSV). Import into InfluxDB 3 using Line Protocol writes. For the export step, follow the instructions in the section called "Exporting Timestream data to Amazon S3". Bulk import from Parquet files is on the roadmap for a future release.
6. **Update monitoring:** Replace LiveAnalytics CloudWatch metrics with InfluxDB 3 CloudWatch metrics (`CPUUtilization`, `MemoryUtilization`, `S3ObjectStorageBytes`). Additional metrics are available via the `/metrics` Prometheus endpoint on the cluster.
7. **Update networking and auth:** LiveAnalytics uses IAM for data-plane auth. InfluxDB 3 uses **token-based auth** (Admin Token created at cluster provisioning). IAM-based data-plane auth for InfluxDB 3 is not yet available. The cluster must be deployed in a VPC with appropriate security groups.

Additional resources

- Getting started with InfluxDB 3
- InfluxDB 3 DB Clusters
- InfluxDB 3 Processing Engine
- Line Protocol Reference
- SQL Reference for InfluxDB 3

Data ingestion into Timestream for InfluxDB

The InfluxDB ingestion script ingests compressed line protocol datasets to Timestream for InfluxDB. A directory containing gzip compressed line protocol files is passed in as a command line argument along with the ingestion destination InfluxDB bucket. This script was designed to ingest multiple files at a time using multi-processing to utilize the resources with InfluxDB and the machine executing the script.

The script does following:

- Extracts zipped files and ingests them into InfluxDB.
- Implements retry mechanisms and error handling.
- Tracks successful and failed ingestions for resuming.
- Optimizes I/O operations when reading from line protocol dataset.

Prerequisites and installation

See the Prerequisites and Installation section in the ingestion script's README in GitHub.

Data preparation

The zipped line protocol files required for ingestion are generated by the data transform scripts. Follow these steps to prepare your data:

1. Set up an EC2 instance with sufficient storage to hold the transformed dataset.
2. Sync the transformed data from the S3 bucket to your local directory:

```
aws s3 sync \
    s3://your-bucket-name/path/to/transformed/data \
    ./data_directory
```

3. Make sure you have read access to all files in the data directory.
4. Run the following ingestion script to ingest data into Timestream for InfluxDB.

Usage

```
python influxdb_ingestion.py <bucket_name> <data_directory> [options]
```

Basic usage

```
python influxdb_ingestion.py my_bucket ./data_files
```

Ingestion rates

We have run some tests for Ingestion rates. Ingestion tests using a C5N.9XL EC2 instance executing the ingestion script with 10 Workers, and ingesting ~500 GB line protocol to 8XL Timestream for InfluxDB instances:

- 3K IOPS 15.86 GB/hour.
- 12K IOPS 70.34 GB/hour.
- 16K IOPS 71.28 GB/hour.

Recommendations

- Use an EC2 instance with sufficient CPU cores to handle parallel processing.
- Ensure the instance has enough storage to hold the entire transformed dataset with additional room for extraction.
  - The number of files extracted at one time is equal to the number of workers configured during script execution.
- Position the EC2 instance in the same region and AZ (if possible) as your InfluxDB instance to minimize latency.
- Consider using instance types optimized for network operations, for example C5N.
- If high ingestion rates are required, at least 12K IOPS is recommended for the Timestream for InfluxDB instance. Additional optimizations can be gained by increasing the worker count for the script dependent on Timestream for InfluxDB instance size.

For more information, see the ingestion script's README.

Migration validation script

The validation script compares logical row/point counts between a source table (Amazon Timestream or Amazon Athena) and an InfluxDB bucket measurement, with optional time-range specifications. This tool helps ensure data integrity during migration processes by running parallel queries against both systems and comparing the results.

The validation script supports queries against either the exported dataset in Athena or the original Timestream database/table. Be aware that querying Timestream directly may lead to inaccurate comparisons if data has been written since the export. The validation script can be run anytime after ingestion has begun. It first polls InfluxDB's metrics endpoint to wait for the WAL (Write-Ahead Log) to flush completely, ensuring all data processing, including post-ingestion file merging and de-duplication, is finished. The script then executes count-only queries over identical time windows, comparing results to highlight matches or mismatches. It supports optional schema/tag filtering for transformed schemas where dimensions are used as fields, and produces human-readable timing and result summaries to facilitate validation of the migration process.

Prerequisites and installation

See the prerequisites and installation section in the Migration Validation Script README.

Usage

```
python validator.py [options]
```

All settings can be supplied as CLI flags or environment variables. See the example.env file within the repository.

For troubleshooting and recommendations see the Migration Validation Script README.

Cleanup

After finishing a migration, following resources/artifacts will be created:

- An Athena table, containing Timestream for LiveAnalytics data. By default, this is <Timestream database name>_<Timestream table name> in the default Athena database.
- An Athena table, containing transformed line protocol data. By default, this is lp_<Athena table name> in the default Athena database.
- Line protocol data within your S3 bucket, with the path <Timestream database name>/<Timestream table name>/unload-<%Y-%m-%d-%H:%M:%S>/line-protocol-output.

- Unloaded data that was created as part of Timestream for LiveAnalytics export tool.
- Downloaded data and logs on your EC2 instance.
- DynamoDB table if used for logging as part of Timestream for LiveAnalytics export tool.

Cleaning up Athena resources

To delete any Athena table, run the following AWS CLI command, replacing <Athena table name> with the name of the table that you want to delete and <Athena database name> with the name of the Athena database that the table resides in:

```
aws glue delete-table \
    --database-name <Athena database name> \
    --name <Athena table name>
```

Cleaning up S3 resources

To delete line protocol data within your S3 bucket, run the following AWS CLI command, replacing <S3 bucket name> with the name of your S3 bucket, <Timestream database name> with the name of your Timestream for LiveAnalytics database, <Timestream table name> with the name of your Timestream for LiveAnalytics table, and <timestamp> with the timestamp that forms the unload-<%Y-%m-%d-%H:%M:%S> path in your S3 bucket:

```
aws s3 rm \
    s3://<S3 bucket name>/<Timestream database name>/<Timestream table name>/unload-
<timestamp>/line-protocol-output \
    --recursive
```

To delete an S3 bucket, run the following command, replacing <S3 bucket name> with the name of your S3 bucket:

```
aws s3 delete-bucket --bucket <S3 bucket name>
```

Cleaning up DynamoDB resources

To delete a DynamoDB table, run the following command, replacing <table name> with the name of the DynamoDB table that you want to delete:

```
aws dynamodb delete-table --table-name <table name>
```

Aurora/RDS Postgres as a target

This section explains ingesting the S3 staged time-series data into Amazon RDS/Aurora PostgreSQL. The ingestion process will primarily focus on the CSV files generated from Timestream's export tool to ingest into Postgres. We recommend designing the PostgreSQL schema and table with proper indexing strategies for time-based queries. Use any ETL processes to transform Timestream's specialized structures into relational tables optimized for your specific requirements. When migrating Timestream data to a relational database, structure your schema with a timestamp column as the primary time index, measurement identifier columns derived from Timestream's measure_name, and dimension columns from Timestream's dimensions and your actual measures. Create strategic indexes on time ranges and frequently queried dimension combinations to optimize performance during data transformation and loading process. When migrating time-series data to PostgreSQL, proper instance sizing is critical for maintaining query performance at scale. Consider your expected data volume, query complexity, and concurrency requirements when selecting an instance class, with particular attention to memory allocation for time-series aggregation workloads. For datasets exceeding tens of millions of rows, leverage PostgreSQL's native partitioning capabilities and advanced indexing strategies to optimize for time-series access patterns.

We recommend performing functional and performance testing to choose the right instance and tuning your PostgreSQL database to address any performance bottlenecks. Performing rigorous data integrity checks through sample query comparisons between your source Timestream database and target system is critical to ensure migration success and maintain query correctness. By executing identical queries against both systems and comparing results — including record counts, aggregations, and outlier values — you can identify any discrepancies that might indicate transformation errors, data loss, or semantic differences in query interpretation. This verification process validates that your data maintains its analytical value post-migration, builds confidence in the new system among stakeholders who rely on these insights, helps identify any necessary query adjustments to accommodate syntax or functional differences between platforms, and establishes a quantifiable baseline for determining when the migration can be considered complete and successful. Without these systematic checks, subtle data inconsistencies might remain undetected, potentially leading to incorrect business decisions or undermining confidence in the entire migration project.

Ingestion

We recommend using AWSDatabase Migration Service (DMS) with source as S3 (both CSV and Parquet are supported) with PostgreSQL as the target. For scenarios where AWS DMS may not

be suitable for your specific requirements, we provide a supplementary Python-based utility (PostgreSQL CSV Ingestion Tool) for migrating CSV data from S3 to PostgreSQL.

Overview of PostgreSQL CSV ingestion tool

The PostgreSQL CSV Ingestion Tool, is a high-performance utility designed to efficiently load CSV files into PostgreSQL databases. It leverages multi-threading and connection pooling to process multiple files in parallel, significantly reducing data loading time. We recommend to running this script using an EC2 instance. Consider using instance types optimized for network operations, such as C5N.

Key features

- Multi-threaded Processing: Loads multiple CSV files simultaneously.
- Connection Pooling: Efficiently manages database connections.
- Automatic Column Detection: Dynamically extracts column names from CSV headers.
- Retry Logic: Handles transient errors with exponential backoff.
- File Management; Moves processed files to a designated directory so retrying is resuming but not restarting.
- Comprehensive Logging: Detailed logs for monitoring and troubleshooting.
- Error Notifications: Optional SNS notifications for failures.
- Secure Credentials: Retrieves database passwords from AWS Secrets Manager.

Prerequisites and installation

See prerequisites and installation in the PostgreSQL CSV Ingestion Tool Readme in GitHub.

Usage

```
python copy_postgres.py \
    --database 'postgres_testing' \
    --table 'demolarge_restored' \
    --csv-files-dir '/data/csv_files/*partition*/*.csv' \
    --host database-1.cluster-xxxxxxxx.us-east-1.rds.amazonaws.com \
    --secret-arn 'arn:aws:secretsmanager:<region>:<account_id>:secret:rds!cluster-
xxxxx-xx-xx-xx-xxxxxxxx-xxxxx' \
    --sns-topic-arn 'arn:aws:sns:<region>:<account_id>:<topic_name>'
```

Validation

You can use DynamoDB for exported rows or logs generated by Timestream's export tool and compare against rows ingested from PostgreSQL Ingestion automation logs. You can do select counts against the source and target tables with the consistent export and import time, if the data is being continuously ingested during the migration process the counts will vary so the recommendation is to compare rows exported and rows important from logging.

Cleanup

- Cleanup unloaded data that was created as part of Timestream for LiveAnalytics export tool.
- Delete downloaded data and logs on EC2 to reclaim the space.
- Delete DynamoDB table if used for logging as part of Timestream for LiveAnalytics export tool.

How it works

The following sections provide an overview of Amazon Timestream for Live Analytics service components and how they interact.

After you read this introduction, see the Accessing Timestream for LiveAnalytics sections to learn how to access Timestream for Live Analytics using the console, AWS CLI, or SDKs.

Topics

- Amazon Timestream for LiveAnalytics concepts
- Architecture
- Writes
- Storage
- Queries
- Scheduled queries
- Timestream Compute Unit (TCU)

Amazon Timestream for LiveAnalytics concepts

Time series data is a sequence of data points recorded over a time interval. This type of data is used for measuring events that change over time. Examples include the following.

- Stock prices over time
- Temperature measurements over time
- CPU utilization of an EC2 instance over time

With time series data, each data point consists of a timestamp, one or more attributes, and the event that changes over time. This data can be used to derive insights into the performance and health of an application, detect anomalies, and identify optimization opportunities. For example, DevOps engineers might want to view data that measures changes in infrastructure performance metrics. Manufacturers might want to track IoT sensor data that measures changes in equipment across a facility. Online marketers might want to analyze clickstream data that captures how a user navigates a website over time. Because time series data is generated from multiple sources in extremely high volumes, it needs to be cost-effectively collected in near real time, and therefore requires efficient storage that helps organize and analyze the data.

Following are the key concepts of Timestream for LiveAnalytics.

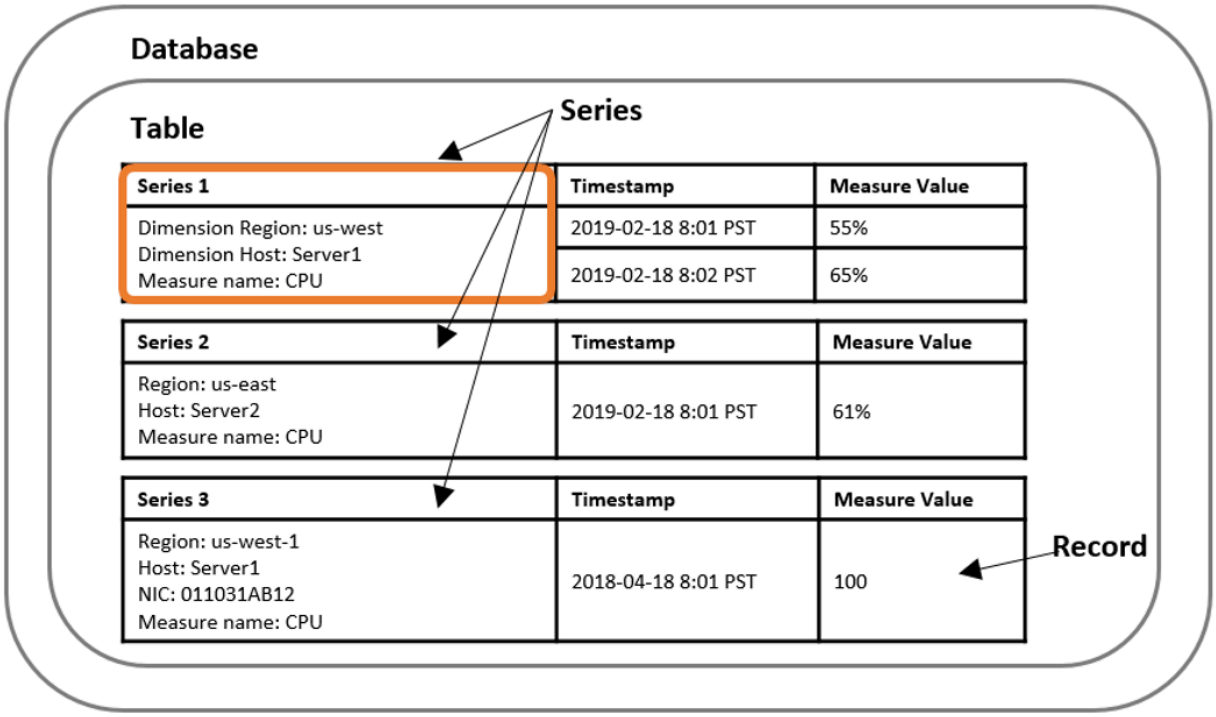
- **Time series** - *A sequence of one or more data points (or records) recorded over a time interval.* Examples are the price of a stock over time, the CPU or memory utilization of an EC2 instance over time, and the temperature/pressure reading of an IoT sensor over time.
- **Record** - *A single data point in a time series.*
- **Dimension** - *An attribute that describes the meta-data of a time series.* A dimension consists of a dimension name and a dimension value. Consider the following examples:
  - When considering a stock exchange as a dimension, the dimension name is "stock exchange" and the dimension value is "NYSE"
  - When considering an AWS Region as a dimension, the dimension name is "region" and the dimension value is "us-east-1"
  - For an IoT sensor, the dimension name is "device ID" and the dimension value is "12345"
- **Measure** - *The actual value being measured by the record.* Examples are the stock price, the CPU or memory utilization, and the temperature or humidity reading. Measures consist of measure names and measure values. Consider the following examples:
  - For a stock price, the measure name is "stock price" and the measure value is the actual stock price at a point in time.
  - For CPU utilization, the measure name is "CPU utilization" and the measure value is the actual CPU utilization.

Measures can be modeled in Timestream for LiveAnalytics as multi-measure or single-measure records. For more information, see Multi-measure records vs. single-measure records.

- **Timestamp** - *Indicates when a measure was collected for a given record.* Timestream for LiveAnalytics supports timestamps with nanosecond granularity.
- **Table** - *A container for a set of related time series.*
- **Database** - *A top level container for tables.*

A summary of Timestream for LiveAnalytics concepts

A **database** contains 0 or more **tables**. Each **table** contains 0 or more **time series**. Each **time series** consists of a sequence of **records** over a given time interval at a specified **granularity**. Each **time series** can be described using its meta-data or **dimensions**, its data or **measures**, and its **timestamps**.



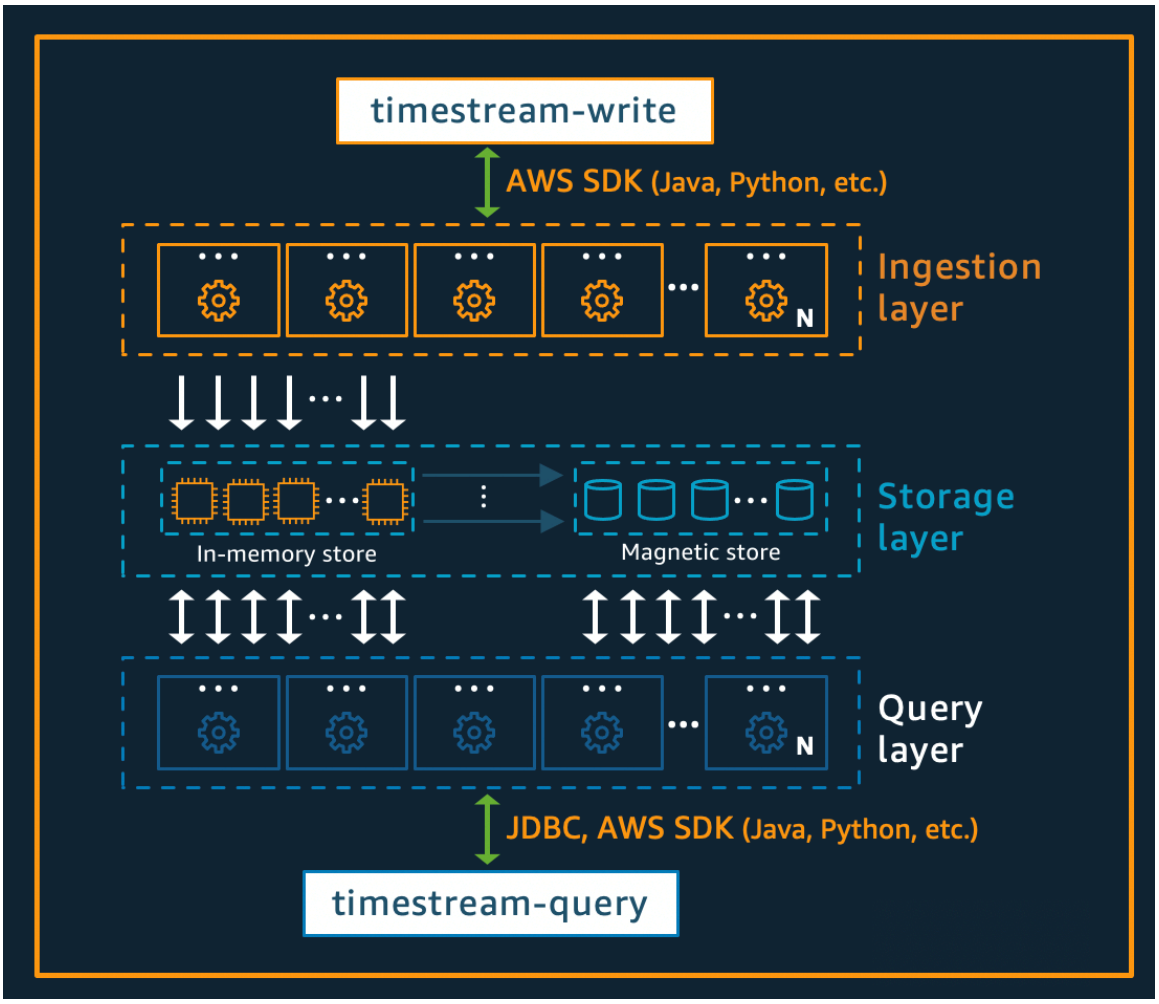
Architecture

Amazon Timestream for Live Analytics has been designed from the ground up to collect, store, and process time series data at scale. Its serverless architecture supports fully decoupled data ingestion, storage, and query processing systems that can scale independently. This design simplifies each subsystem, making it easier to achieve unwavering reliability, eliminate scaling bottlenecks, and

reduce the chances of correlated system failures. Each of these factors becomes more important as the system scales.

Topics

- Write architecture
- Storage architecture
- Query architecture
- Cellular architecture



Write architecture

When writing time-series data, Amazon Timestream for Live Analytics routes writes for a table, partition, to a fault-tolerant memory store instance that processes high throughput data writes. The memory store in turn achieves durability in a separate storage system that replicates the data

across three Availability Zones (AZs). Replication is quorum based such that the loss of nodes, or an entire AZ, will not disrupt write availability. In near real-time, other in-memory storage nodes sync to the data in order to serve queries. The reader replica nodes span AZs as well, to ensure high read availability.

Timestream for Live Analytics supports writing data directly into the magnetic store, for applications generating lower throughput late-arriving data. Late-arriving data is data with a timestamp earlier than the current time. Similar to the high throughput writes in the memory store, the data written into the magnetic store is replicated across three AZs and the replication is quorum based.

Whether data is written to the memory or magnetic store, Timestream for Live Analytics automatically indexes and partitions data before writing it to storage. A single Timestream for Live Analytics table may have hundreds, thousands, or even millions of partitions. Individual partitions do not, directly, communicate with each other and do not share any data (shared-nothing architecture). Instead, the partitioning of a table is tracked through a highly available partition tracking and indexing service. This provides another separation of concerns designed specifically to minimize the effect of failures in the system and make correlated failures much less likely.

Storage architecture

When data is stored in Timestream for Live Analytics, data is organized in time order as well as across time based on context attributes written with the data. Having a partitioning scheme that divides "space" in addition to time is important for massively scaling a time series system. This is because most time series data is written at or around the current time. As a result, partitioning by time alone does not do a good job of distributing write traffic or allowing for effective pruning of data at query time. This is important for extreme scale time series processing, and it has allowed Timestream for Live Analytics to scale orders of magnitude higher than the other leading systems out there today in serverless fashion. The resulting partitions are referred to as "tiles" because they represent divisions of a two-dimensional space (which are designed to be of a similar size). Timestream for Live Analytics tables start out as a single partition (tile), and then split in the spatial dimension as throughput requires. When tiles reach a certain size, they then split in the time dimension in order to achieve better read parallelism as the data size grows.

Timestream for Live Analytics is designed to automatically manage the lifecycle of time series data. Timestream for Live Analytics offers two data stores—an in-memory store and a cost-effective magnetic store. It also supports configuring table-level policies to automatically transfer data

across stores. Incoming high throughput data writes land in the memory store where data is optimized for writes, as well as reads performed around current time for powering dashboard and alerting type queries. When the main time frame for writes, alerting, and dashboarding needs has passed, allowing the data to automatically flow from the memory store to the magnetic store to optimize cost. Timestream for Live Analytics allows setting a data retention policy on the memory store for this purpose. Data writes for late-arriving data are directly written into the magnetic store.

Once the data is available in the magnetic store (because of expiration of the memory store retention period or because of direct writes into the magnetic store), it is reorganized into a format that is highly optimized for large volume data reads. The magnetic store also has a data retention policy that may be configured if there is a time threshold where the data outlives its usefulness. When the data exceeds the time range defined for the magnetic store retention policy, it is automatically removed. Therefore, with Timestream for Live Analytics, other than some configuration, the data lifecycle management occurs seamlessly behind the scenes.

Query architecture

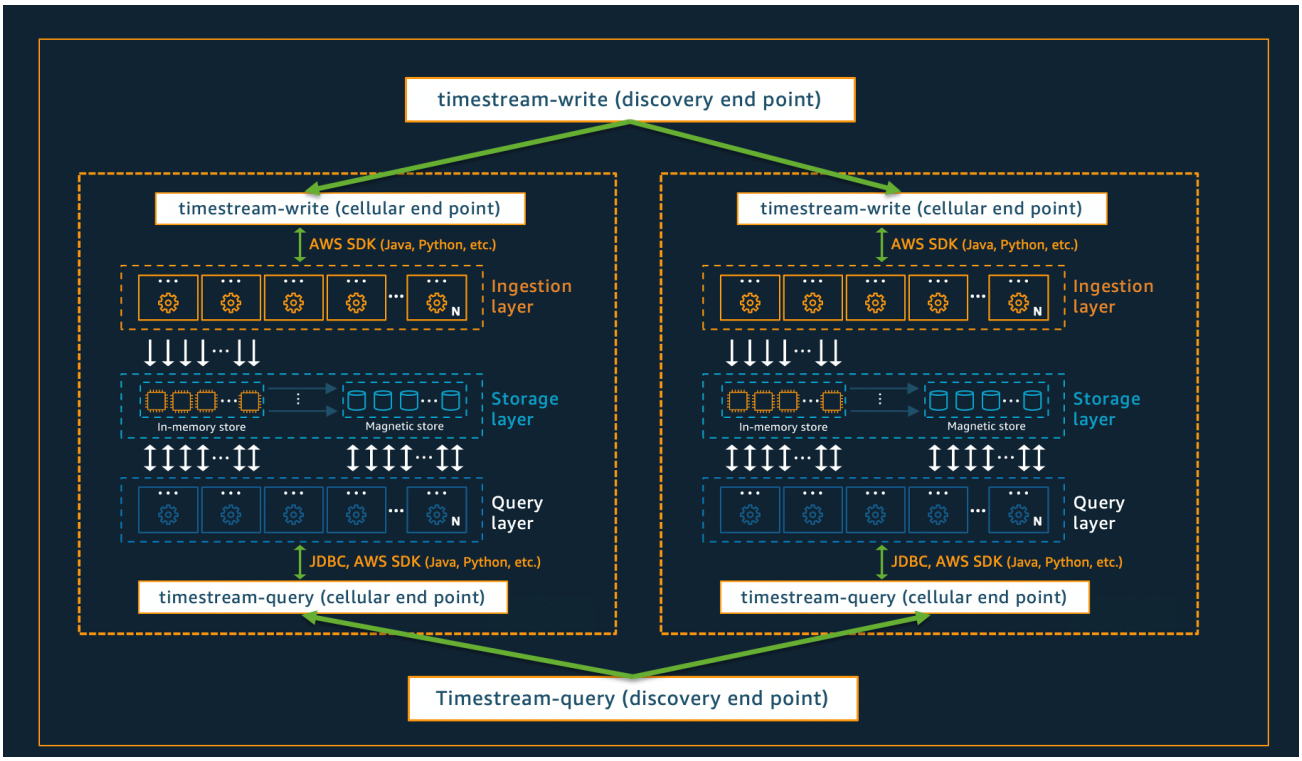
Timestream for Live Analytics queries are expressed in a SQL grammar that has extensions for time series-specific support (time series-specific data types and functions), so the learning curve is easy for developers already familiar with SQL. Queries are then processed by an adaptive, distributed query engine that uses metadata from the tile tracking and indexing service to seamlessly access and combine data across data stores at the time the query is issued. This makes for an experience that resonates well with customers as it collapses many of the Rube Goldberg complexities into a simple and familiar database abstraction.

Queries are run by a dedicated fleet of workers where the number of workers enlisted to run a given query is determined by query complexity and data size. Performance for complex queries over large datasets is achieved through massive parallelism, both on the query runtime fleet and the storage fleets of the system. The ability to analyze massive amounts of data quickly and efficiently is one of the greatest strengths of Timestream for Live Analytics. A single query that runs over terabytes or even petabytes of data might have thousands of machines working on it all at the same time.

Cellular architecture

To ensure that Timestream for Live Analytics can offer virtually infinite scale for your applications, while simultaneously ensuring 99.99% availability, the system is also designed using a cellular

architecture. Rather than scaling the system as a whole, Timestream for Live Analytics segments into multiple smaller copies of itself, referred to as *cells*. This allows cells to be tested at full scale, and prevents a system problem in one cell from affecting activity in any other cells in a given region. While Timestream for Live Analytics is designed to support multiple cells per region, consider the following fictitious scenario, in which there are 2 cells in a region.



In the scenario depicted above, the data ingestion and query requests are first processed by the discovery endpoint for data ingestion and query, respectively. Then, the discovery endpoint identifies the cell containing the customer data, and directs the request to the appropriate ingestion or query endpoint for that cell. When using the SDKs, these endpoint management tasks are transparently handled for you.

> **Note**
>
> *When using VPC endpoints with Timestream for Live Analytics or directly accessing REST API operations for Timestream for Live Analytics, you will need to interact directly with the cellular endpoints.* For guidance on how to do so, see VPC Endpoints for instructions on how to set up VPC endpoints, and Endpoint Discovery Pattern for instructions on direct invocation of the REST API operations.

Writes

You can collect time series data from connected devices, IT systems, and industrial equipment, and write it into Timestream for Live Analytics. Timestream for Live Analytics enables you to write data points from a single time series and/or data points from many series in a single write request when the time series belong to the same table. For your convenience, Timestream for Live Analytics offers you with a flexible schema that auto detects the column names and data types for your Timestream for Live Analytics tables based on the dimension names and the data types of the measure values you specify when invoking writes into the database. You can also write batches of data into Timestream for Live Analytics.

> **Note**
>
> Timestream for Live Analytics supports eventual consistency semantics for reads. This means that when you query data immediately after writing a batch of data into Timestream for Live Analytics, the query results might not reflect the results of a recently completed write operation. The results may also include some stale data. Similarly, while writing time series data with one or more new dimensions, a query can return a partial subset of columns for a short period of time. If you repeat these query requests after a short time, the results should return the latest data.

You can write data using the AWS SDKs, AWS CLI, or through AWS Lambda, AWS IoT Core, Amazon Managed Service for Apache Flink, Amazon Kinesis, Amazon MSK, and Open source Telegraf.

Topics

- Data types
- No upfront schema definition
- Writing data (inserts and upserts)
- Eventual consistency for reads
- Batching writes with WriteRecords API
- Batch load
- Choosing between the WriteRecords API operation and batch load

Data types

Timestream for Live Analytics supports the following data types for writes.

Data type	Description
BIGINT	Represents a 64-bit signed integer.
BOOLEAN	Represents the two truth values of logic, namely, true, and false.
DOUBLE	64-bit variable-precision implementing the IEEE Standard 754 for Binary Floating-Point Arithmetic. **Note** There are query language functions for `Infinity` and `NaN` double values which can be used in queries. But you cannot write those values to Timestream.
VARCHAR	Variable length character data with an optional maximum length. The maximum limit is 2 KB.
MULTI	Data type for multi-measure records. This data type includes one or more measures of type `BIGINT`, `BOOLEAN`, `DOUBLE`, `VARCHAR`, and `TIMESTAMP` .
TIMESTAMP	Represents an instance in time using nanosecond precision time in UTC, tracking the time since Unix time. This data type is currently supported only for multi-measure records (i.e. within measure values of type `MULTI`). `YYYY-MM-DD hh:mm:ss.sssssssss` Writes support timestamps in the range `1970-01-01 00:00:00.000000000` to `2262-04-11 23:47:16.854775807` .

No upfront schema definition

Before sending data into Amazon Timestream for Live Analytics, you must create a database and a table using the AWS Management Console, Timestream for Live Analytics SDKs, or the Timestream for Live Analytics API operations. For more information, see Create a database and Create a table. While creating the table, you do not need to define the schema up front. Amazon Timestream for Live Analytics automatically detects the schema based on the measures and dimensions of the data points being sent, so you no longer need to alter your schema offline to adapt it to your rapidly changing time series data.

Writing data (inserts and upserts)

The write operation in Amazon Timestream for Live Analytics enables you to insert and *upsert* data. By default, writes in Amazon Timestream for Live Analytics follow the *first writer wins* semantics, where data is stored as append only and duplicate records are rejected. While the first writer wins semantics satisfies the requirements of many time series applications, there are scenarios where applications need to update existing records in an idempotent manner and/or write data with the last writer wins semantics, where the record with the highest version is stored in the service. To address these scenarios, Amazon Timestream for Live Analytics provides the ability to upsert data. Upsert is an operation that inserts a record into the system when the record does not exist, or updates the record when one exists. When the record is updated, it is updated in an idempotent manner.

There isn't a record level operation for deletion. But tables and databases can be deleted.

Writing data into the memory store and the magnetic store

Amazon Timestream for Live Analytics offers the ability to directly write data into the memory store and the magnetic store. The memory store is optimized for high throughput data writes and the magnetic store is optimized for lower throughput writes of late arrival data.

Late-arriving data is data with a timestamp earlier than the current time and outside the memory store retention period. You must explicitly enable the ability to write late-arriving data into the magnetic store by enabling magnetic store writes for the table. Also, `MagneticStoreRejectedDataLocation` is defined when a table is created. To write to the magnetic store, callers of `WriteRecords` must have `S3:PutObject` permissions to the S3 bucket specified in `MagneticStoreRejectedDataLocation`during table creation. For more information, see CreateTable, WriteRecords, and PutObject.

Writing data with single-measure records and multi-measure records

Amazon Timestream for Live Analytics offers the ability to write data using two types of records, namely, single-measure records and multi-measure records.

Single-measure records

Single-measure records enable you to send a single measure per record. When data is sent to Timestream for Live Analytics using this format, Timestream for Live Analytics creates one table row per record. This means that if a device emits 4 metrics and each metric is sent as a single-measure record, Timestream for Live Analytics will create 4 rows in the table to store this data, and the device attributes will be repeated for each row. This format is recommended in cases when you want to monitor a single metric from an application or when your application does not emit multiple metrics at the same time.

Multi-measure records

With multi-measure records, you can store multiple measures in a single table row, instead of storing one measure per table row. Multi-measure records therefore enable you to migrate your existing data from relational databases to Amazon Timestream for Live Analytics with minimal changes.

You can also batch more data in a single write request than single-measure records. This increases data write throughput and performance, and also reduces the cost of data writes. This is because batching more data in a write request enables Amazon Timestream for Live Analytics to identify more repeatable data in a single write request (where applicable), and charge only once for repeated data.

Topics

- Multi-measure records
- Writing data with a timestamp that exists in the past or in the future

Multi-measure records

With multi-measure records, you can store your time-series data in a more compact format in the memory and magnetic store, which helps lower data storage costs. Also, the compact data storage lends itself to writing simpler queries for data retrieval, improves query performance, and lowers the cost of queries.

Furthermore, multi-measure records also support the TIMESTAMP data type for storing more than one timestamp in a time-series record. TIMESTAMP attributes in a multi-measure record support timestamps in future or past. Multi-measure records therefore help improve performance, cost, and query simplicity—and offer more flexibility for storing different types of correlated measures.

Benefits

The following are the benefits of using multi-measure records.

- **Performance and cost** – Multi-measure records enable you to write multiple time-series measures in a single write request. This increases the write throughput and also reduces the cost of writes. With multi-measure records, you can store data in a more compact manner, which helps lower the data storage costs. The compact data storage of multi-measure records results in less data being processed by queries. This is designed to improve the overall query performance and help lower the query cost.
- **Query simplicity** – With multi-measure records, you do not need to write complex common table expressions (CTEs) in a query to read multiple measures with the same timestamp. This is because the measures are stored as columns in a single table row. Multi-measure records therefore enable writing simpler queries.
- **Data modeling flexibility** – You can write future timestamps into Timestream for Live Analytics by using the TIMESTAMP data type and multi-measure records. A multi-measure record can have multiple attributes of TIMESTAMP data type, in addition to the time field in a record. TIMESTAMP attributes, in a multi-measure record, can have timestamps in the future or the past and behave like the time field except that Timestream for Live Analytics does not index on the values of type TIMESTAMP in a multi-measure record.

Use cases

You can use multi-measure records for any time-series application that generates more than one measurement from the same device at any given time. The following are some example applications.

- A video streaming platform that generates hundreds of metrics at a given time.
- Medical devices that generate measurements such as blood oxygen levels, heart rate, and pulse.
- Industrial equipment such as oil rigs that generate metrics, temperature, and weather sensors.
- Other applications that are architected with one or more microservices.

Example: Monitoring the performance and health of a video streaming application

Consider a video streaming application that is running on 200 EC2 instances. You want to use Amazon Timestream for Live Analytics to store and analyze the metrics being emitted from the application, so you can understand the performance and health of your application, quickly identify anomalies, resolve issues, and discover optimization opportunities.

We will model this scenario with single-measure records and multi-measure records, and then compare/contrast both approaches. For each approach, we make the following assumptions.

- Each EC2 instance emits four measures (video_startup_time, rebuffering_ratio, video_playback_failures, and average_frame_rate) and four dimensions (device_id, device_type, os_version, and region) per second.
- You want to store 6 hours of data in the memory store and 6 months of data in the magnetic store.
- To identify anomalies, you've set up 10 queries that run every minute to identify any unusual activity over the past few minutes. You've also built a dashboard with eight widgets that display the last 6 hours of data, so that you can effectively monitor your application. This dashboard is accessed by five users at any given time and is auto-refreshed every hour.

Using single measure records

Data modeling: With single measure records, we will create one record for each of the four measures (video startup time, rebuffering ratio, video playback failures, and average frame rate). Each record will have the four dimensions (device_id, device_type, os_version, and region) and a timestamp.

Writes: When you write data into Amazon Timestream for Live Analytics, the records are constructed as follows.

```
public void writeRecords() {
    System.out.println("Writing records");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();

    final Dimension device_id = new
 Dimension().withName("device_id").withValue("12345678");
```

```
    final Dimension device_type = new
 Dimension().withName("device_type").withValue("iPhone 11");
    final Dimension os_version = new
 Dimension().withName("os_version").withValue("14.8");
    final Dimension region = new Dimension().withName("region").withValue("us-east-1");

    dimensions.add(device_id);
    dimensions.add(device_type);
    dimensions.add(os_version);
    dimensions.add(region);

    Record videoStartupTime = new Record()
        .withDimensions(dimensions)
        .withMeasureName("video_startup_time")
        .withMeasureValue("200")
        .withMeasureValueType(MeasureValueType.BIGINT)
        .withTime(String.valueOf(time));
    Record rebufferingRatio = new Record()
        .withDimensions(dimensions)
        .withMeasureName("rebuffering_ratio")
        .withMeasureValue("0.5")
        .withMeasureValueType(MeasureValueType.DOUBLE)
        .withTime(String.valueOf(time));
    Record videoPlaybackFailures = new Record()
        .withDimensions(dimensions)
        .withMeasureName("video_playback_failures")
        .withMeasureValue("0")
        .withMeasureValueType(MeasureValueType.BIGINT)
        .withTime(String.valueOf(time));
    Record averageFrameRate = new Record()
        .withDimensions(dimensions)
        .withMeasureName("average_frame_rate")
        .withMeasureValue("0.5")
        .withMeasureValueType(MeasureValueType.DOUBLE)
        .withTime(String.valueOf(time));

    records.add(videoStartupTime);
    records.add(rebufferingRatio);
    records.add(videoPlaybackFailures);
    records.add(averageFrameRate);

    WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
        .withDatabaseName(DATABASE_NAME)
        .withTableName(TABLE_NAME)
```

```
        .withRecords(records);

    try {
      WriteRecordsResult writeRecordsResult =
 amazonTimestreamWrite.writeRecords(writeRecordsRequest);
      System.out.println("WriteRecords Status: " +
 writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("RejectedRecords: " + e);
      for (RejectedRecord rejectedRecord : e.getRejectedRecords()) {
        System.out.println("Rejected Index " + rejectedRecord.getRecordIndex() + ": "
            + rejectedRecord.getReason());
      }
      System.out.println("Other records were written successfully. ");
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

When you store single-measure records, the data is logically represented as follows.

Time	device_id	device_type	os_version	region	measure_name	measure_value::bigint	measure_value::double
2021-09-07 21:48:44 .00	12345678	iPhone 11	14.8	us-east-1	video_startup_time	200	
2021-09-07 21:48:44 .00	12345678	iPhone 11	14.8	us-east-1	rebuffering_ratio		0.5
2021-09-07 21:48:44 .00	12345678	iPhone 11	14.8	us-east-1	video_playback_failures	0	

Time	device_id	device_type	os_version	region	measure_name	measure_value::bigint	measure_value::double
2021-09-07 21:48:44 .00	12345678	iPhone 11	14.8	us-east-1	average_frame_rate		0.85
2021-09-07 21:53:44 .00	12345678	iPhone 11	14.8	us-east-1	video_startup_time	500	
2021-09-07 21:53:44 .00	12345678	iPhone 11	14.8	us-east-1	rebuffering_ratio		1.5
2021-09-07 21:53:44 .00	12345678	iPhone 11	14.8	us-east-1	video_playback_failures	10	
2021-09-07 21:53:44 .00	12345678	iPhone 11	14.8	us-east-1	average_frame_rate		0.2

Queries: You can write a query that retrieves all of the data points with the same timestamp received over the past 15 minutes as follows.

```
with cte_video_startup_time as ( SELECT time, device_id, device_type, os_version,
 region, measure_value::bigint as video_startup_time FROM table where time >= ago(15m)
 and measure_name="video_startup_time"),
cte_rebuffering_ratio as ( SELECT time, device_id, device_type, os_version, region,
 measure_value::double as rebuffering_ratio FROM table where time >= ago(15m) and
 measure_name="rebuffering_ratio"),
```

```
cte_video_playback_failures as ( SELECT time, device_id, device_type, os_version,
 region, measure_value::bigint as video_playback_failures FROM table where time >=
 ago(15m) and measure_name="video_playback_failures"),
cte_average_frame_rate as ( SELECT time, device_id, device_type, os_version, region,
 measure_value::double as average_frame_rate FROM table where time >= ago(15m) and
 measure_name="average_frame_rate")
SELECT a.time, a.device_id, a.os_version, a.region, a.video_startup_time,
 b.rebuffering_ratio, c.video_playback_failures, d.average_frame_rate FROM
 cte_video_startup_time a, cte_buffering_ratio b, cte_video_playback_failures c,
 cte_average_frame_rate d WHERE
a.time = b.time AND a.device_id = b.device_id AND a.os_version = b.os_version AND
 a.region=b.region AND
a.time = c.time AND a.device_id = c.device_id AND a.os_version = c.os_version AND
 a.region=c.region AND
a.time = d.time AND a.device_id = d.device_id AND a.os_version = d.os_version AND
 a.region=d.region
```

Workload cost: The cost of this workload is estimated to be $373.23 per month with single-measure records

Using multi-measure records

Data modeling: With multi-measure records, we will create one record that contains all four measures (video startup time, rebuffering ratio, video playback failures, and average frame rate), all four dimensions (device_id, device_type, os_version, and region), and a timestamp.

Writes: When you write data into Amazon Timestream for Live Analytics, the records are constructed as follows.

```
public void writeRecords() {
    System.out.println("Writing records");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();

    final Dimension device_id = new
 Dimension().withName("device_id").withValue("12345678");
    final Dimension device_type = new
 Dimension().withName("device_type").withValue("iPhone 11");
    final Dimension os_version = new
 Dimension().withName("os_version").withValue("14.8");
```

```
    final Dimension region = new Dimension().withName("region").withValue("us-east-1");

    dimensions.add(device_id);
    dimensions.add(device_type);
    dimensions.add(os_version);
    dimensions.add(region);

    Record videoMetrics = new Record()
        .withDimensions(dimensions)
        .withMeasureName("video_metrics")
        .withTime(String.valueOf(time));
        .withMeasureValueType(MeasureValueType.MULTI)
        .withMeasureValues(
          new MeasureValue()
         .withName("video_startup_time")
         .withValue("0")
         .withValueType(MeasureValueType.BIGINT),
         new MeasureValue()
  .withName("rebuffering_ratio")
         .withValue("0.5")
         .withType(MeasureValueType.DOUBLE),
          new MeasureValue()
         .withName("video_playback_failures")
         .withValue("0")
         .withValueType(MeasureValueType.BIGINT),
   new MeasureValue()
          .withName("average_frame_rate")
         .withValue("0.5")
         .withValueType(MeasureValueType.DOUBLE))

    records.add(videoMetrics);

    WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
        .withDatabaseName(DATABASE_NAME)
        .withTableName(TABLE_NAME)
        .withRecords(records);

    try {
      WriteRecordsResult writeRecordsResult =
 amazonTimestreamWrite.writeRecords(writeRecordsRequest);
      System.out.println("WriteRecords Status: " +
 writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("RejectedRecords: " + e);
```

```
      for (RejectedRecord rejectedRecord : e.getRejectedRecords()) {
        System.out.println("Rejected Index " + rejectedRecord.getRecordIndex() + ": "
            + rejectedRecord.getReason());
      }
      System.out.println("Other records were written successfully. ");
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

When you store multi-measure records, the data is logically represented as follows.

Time	device_i	device_t pe	os_versi n	region	measure ame	video_st rtup_tim	rebuffer ng_ratio	video_ playbacl failures	average_f rame_rate
2021-09 7 21:48:44 0	1234567	iPhone 11	14.8	us-east-1	video_m rics	200	0.5	0	0.85
2021-09 7 21:53:44 0	1234567	iPhone 11	14.8	us-east-1	video_m rics	500	1.5	10	0.2

Queries: You can write a query that retrieves all of the data points with the same timestamp received over the past 15 minutes as follows.

```
SELECT time, device_id, device_type, os_version, region, video_startup_time,
 rebuffering_ratio, video_playback_failures, average_frame_rate FROM table where time
 >= ago(15m)
```

Workload cost: The cost of workload is estimated to be $127.43 with multi-measure records.

> **Note**
>
> In this case, using multi-measure records reduces the overall estimated monthly spend by 2.5x, with the data writes cost reduced by 3.3x, the storage cost reduced by 3.3x, and the query cost reduced by 1.2x.

Writing data with a timestamp that exists in the past or in the future

Timestream for Live Analytics offers the ability to write data with a timestamp that lies outside of the memory store retention window through a couple different mechanisms.

- **Magnetic store writes** – You can write late-arriving data directly into the magnetic store through magnetic store writes. To use magnetic store writes, you must first enable magnetic store writes for a table. You can then ingest data into the table using the same mechanism used for writing data into the memory store. Amazon Timestream for Live Analytics will automatically write the data into the magnetic store based on its timestamp.

  > **Note**
  >
  > The write-to-read latency for the magnetic store can be up to 6 hours, unlike writing data into the memory store, where the write-to-read latency is in the sub-second range.

- **TIMESTAMP data type for measures** – You can use the TIMESTAMP data type to store data from the past, present, or future. A multi-measure record can have multiple attributes of TIMESTAMP data type, in addition to the time field in a record. TIMESTAMP attributes, in a multi-measure record, can have timestamps in the future or the past and behave like the time field except that Timestream for Live Analytics does not index on the values of type TIMESTAMP in a multi-measure record.

  > **Note**
  >
  > The TIMESTAMP data type is supported only for multi-measure records.

Eventual consistency for reads

Timestream for Live Analytics supports eventual consistency semantics for reads. This means that when you query data immediately after writing a batch of data into Timestream for Live Analytics,

the query results might not reflect the results of a recently completed write operation. If you repeat these query requests after a short time, the results should return the latest data.

Batching writes with WriteRecords API

Amazon Timestream for Live Analytics enables you to write data points from a single time series and/or data points from many series in a single write request. Batching multiple data points in a single write operation is beneficial from a performance and cost perspective. See Writes in the Metering and Pricing section for more details.

> **Note**
>
> Your write requests to Timestream for Live Analytics may be throttled as Timestream for Live Analytics scales to adapt to the data ingestion needs of your application. If your applications encounter throttling exceptions, you must continue to send data at the same (or higher) throughput to allow Timestream for Live Analytics to automatically scale to your application's needs.

Batch load

With *batch load* for Amazon Timestream for LiveAnalytics, you can ingest CSV files stored in Amazon S3 into Timestream in batches. With this new functionality, you can have your data in Timestream for LiveAnalytics without having to rely on other tools or write custom code. You can use batch load for backfilling data with flexible wait times, such as data that isn't immediately required for querying or analysis.

You can create batch load tasks by using the AWS Management Console, the AWS CLI, and the AWS SDKs. For more information, see Using batch load with the console, Using batch load with the AWS CLI, and Using batch load with the AWS SDKs.

For more information about batch load, see Using batch load in Timestream for LiveAnalytics.

Choosing between the WriteRecords API operation and batch load

With the WriteRecords API operation, you can write your streaming time series data into Timestream for LiveAnalytics as it's generated by your system. By using WriteRecords, you can continuously ingest a single data point or smaller batches of data in real time. Timestream for LiveAnalytics offers you a flexible schema that auto detects the column names and data types for

your Timestream for LiveAnalytics tables, based on the dimension names and data types of the data points you specify when invoking writes into the database.

In contrast, *batch load* enables the robust ingestion of batched time-series data from source files (CSV files) into Timestream for LiveAnalytics, using a data model that you define. A few examples for when to use batch load with a source file are importing time series data in bulk for the evaluation of Timestream for LiveAnalytics through a proof of concept, importing time series data in bulk from an IoT device that was offline for some time, and migrating historical time series data from Amazon S3 to Timestream for LiveAnalytics. For information about batch load, see Using batch load in Timestream for LiveAnalytics.

Both solutions are secure, reliable, and performant.

Use WriteRecords when:

- Streaming smaller amounts (less than 10 MB) of data per request.
- Populating existing tables.
- Ingesting data from a log stream.
- Performing real-time analytics.
- Requiring lower latency.

Use batch load when:

- Ingesting larger loads of data that originate in Amazon S3 in CSV files. For more information about limits, see Quotas.
- Populating new tables, such as in the case of a data migration.
- Enriching databases with historical data (ingestion into new tables).
- You have source data that changes slowly or not at all.
- You have flexible wait times because a batch load task might be in a pending state until resources are available, especially if you load a very large amount of data. Batch load is suitable for data that doesn't need to be readily available for querying or analysis to add more clarity.

Storage

Timestream for Live Analytics stores and organizes your time series data to optimize query processing time and to reduce storage costs. It offers data storage tiering and supports two storage

tiers: a memory store and a magnetic store. The memory store is optimized for high throughput data writes and fast point-in-time queries. The magnetic store is optimized for lower throughput late-arriving data writes, long term data storage, and fast analytical queries.

Timestream for Live Analytics ensures durability of your data by automatically replicating your memory and magnetic store data across different Availability Zones within a single AWS Region. All of your data is written to disk before acknowledging your write request as complete.

Timestream for Live Analytics enables you to configure retention policies to move data from the memory store to the magnetic store. When the data reaches the configured value, Timestream for Live Analytics automatically moves the data to the magnetic store. You can also set a retention value on the magnetic store. When data expires out of the magnetic store, it is permanently deleted.

For example, consider a scenario where you configure the memory store to hold a week's-worth of data and the magnetic store to hold 1 year's-worth of data. The age of the data is computed using the timestamp associated with the data point. When the data in the memory store becomes a week old it is automatically moved to the magnetic store. It is then retained in the magnetic store for a year. When the data becomes a year old, it is deleted from Timestream for Live Analytics. The retention values of the memory store and the magnetic store cumulatively define the amount of time that your data will be stored in Timestream for Live Analytics. This means that for the above scenario, from the time of data arrival, the data is stored in Timestream for Live Analytics for a total period of 1 year and 1 week.

> **Note**
>
> When you upgrade the retention period of the memory or magnetic store, the retention change takes effect from that point onwards. For example, if the retention period of the memory store was set to 2 hours and then changed to 24 hours by updating the table retention policies, the memory store will be capable of holding 24 hours of data, but will be populated with 24 hours of data 22 hours after this change was made. Timestream for Live Analytics does not retrieve data from the magnetic store to populate the memory store.

To ensure the security of your time series data, your data in Timestream for Live Analytics is always encrypted by default. This applies to data in transit and at rest. Furthermore, Timestream for Live Analytics enables you to use customer managed keys to secure your data in the magnetic store. For more information on customer managed keys, see AWS KMS keys.

Queries

With Timestream for Live Analytics, you can easily store and analyze metrics for DevOps, sensor data for IoT applications, and industrial telemetry data for equipment maintenance, as well as many other use cases. The purpose-built, adaptive query engine in Timestream for Live Analytics allows you to access data across storage tiers using a single SQL statement. It transparently accesses and combines data across storage tiers without requiring you to specify the data location. You can use SQL to query data in Timestream for Live Analytics to retrieve time series data from one or more tables. You can access the metadata information for databases and tables. Timestream for Live Analytics SQL also supports built-in functions for time series analytics. You can refer to the Query language reference reference for additional details.

Timestream for Live Analytics is designed to have a fully decoupled data ingestion, storage, and query architecture where each component can scale independently of other components (allowing it to offer virtually infinite scale for an application's needs). This means that Timestream for Live Analytics does not "tip over" when your applications send hundreds of terabytes of data per day or run millions of queries processing small or large amounts of data. As your data grows over time, the query latency in Timestream for Live Analytics remains mostly unchanged. This is because the Timestream for Live Analytics query architecture can leverage massive amounts of parallelism to process larger data volumes and automatically scale to match query throughput needs of an application.

Data model

Timestream supports two data models for queries—the flat model and the time series model.

> **Note**
>
> Data in Timestream is stored using the flat model and it is the default model for querying data. The time series model is a query-time concept and is used for time series analytics.

- Flat model
- Time series model

Flat model

The flat model is Timestream's default data model for queries. It represents time series data in a tabular format. The dimension names, time, measure names and measure values appear as columns. Each row in the table is an atomic data point corresponding to a measurement at a specific time within a time series. Timestream databases, tables, and columns have some naming constraints. Those are described in Service limits.

The table below shows an illustrative example for how Timestream stores data representing the CPU utilization, memory utilization, and network activity of EC2 instances, when the data is sent as a single-measure record. In this case, the dimensions are the region, availability zone, virtual private cloud, and instance IDs of the EC2 instances. The measures are the CPU utilization, memory utilization, and the incoming network data for the EC2 instances. The columns region, az, vpc, and instance_id contain the dimension values. The column time contains the timestamp for each record. The column measure_name contains the names of the measures represented by cpu-utilization, memory_utilization, and network_bytes_in. The columns measure_value::double contains measurements emitted as doubles (e.g. CPU utilization and memory utilization). The column measure_value::bigint contains measurements emitted as integers e.g. the incoming network data.

Time	region	az	vpc	instance_id	measure_name	measure_value::double	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	35.0	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	38.2	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	45.3	null

Time	region	az	vpc	instance_id	measure_name	measure_value::double	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	54.9	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	42.6	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	33.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes	34,400	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes	1,500	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes	6,000	null

The table below shows an illustrative example for how Timestream stores data representing the CPU utilization, memory utilization, and network activity of EC2 instances, when the data is sent as a multi-measure record.

Time	region	az	vpc	instance_id	measure_ame	cpu_utilization	memory_ilization	network_bytes
2019-12-4 19:00:00.0000000	us-east-1	us-east-1d	vpc-1a2b c4d	i-1234567 890abcde0	metrics	35.0	54.9	34,400
2019-12-4 19:00:01.0000000	us-east-1	us-east-1d	vpc-1a2b c4d	i-1234567 890abcde0	metrics	38.2	42.6	1,500
2019-12-4 19:00:02.0000000	us-east-1	us-east-1d	vpc-1a2b c4d	i-1234567 890abcde0	metrics	45.3	33.3	6,600

Time series model

The time series model is a query time construct used for time series analytics. It represents data as an ordered sequence of (time, measure value) pairs. Timestream supports time series functions such as interpolation to enable you to fill the gaps in your data. To use these functions, you must convert your data into the time series model using functions such as create_time_series. Refer to Query language reference for more details.

Using the earlier example of the EC2 instance, here is the CPU utilization data expressed as a timeseries.

region	az	vpc	instance_id	cpu_utilization
us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567 890abcdef0	[{time: 2019-12-0 4 19:00:00. 000000000, value: 35}, {time: 2019-12-0

region	az	vpc	instance_id	cpu_utilization
				4 19:00:01. 000000000 , value: 38.2}, {time: 2019-12-0 4 19:00:02. 000000000, value: 45.3}]

Scheduled queries

The scheduled query feature in Amazon Timestream for Live Analytics is a fully managed, serverless, and scalable solution for calculating and storing aggregates, rollups, and other forms of preprocessed data typically used to power operational dashboards, business reports, ad hoc analytics, and other applications. Scheduled queries make real-time analytics more performant and cost-effective, so you can derive additional insights from your data, and can continue to make better business decisions.

For more information about scheduled query, see Using scheduled queries in Timestream for LiveAnalytics.

Timestream Compute Unit (TCU)

Amazon Timestream for Live Analytics measures the compute capacity allocated to you for your query needs in Timestream compute unit (TCU). One TCU comprises of 4 vCPUs and 16 GB of memory. When you run queries in Timestream for Live Analytics, the service allocates TCUs on-demand based on the complexity of your queries and the amount of data being processed. The number of TCUs that a query consumes determines the associated cost.

> **Note**
>
> All AWS accounts that onboard to the service after April 29, 2024 will default to using TCUs for query pricing.

In this topic:

- Provisioned Timestream Compute Units
- MaxQuery TCU
- Billing for TCU
- Configuring TCU
- Estimating required compute units
- When to increase MaxQueryTCU
- When to decrease MaxQueryTCU
- Monitoring usage with CloudWatch metrics
- Understanding variations in compute units usage

Provisioned Timestream Compute Units

> **Note**
>
> Provisioned TCU is available only in the Asia Pacific (Mumbai) region.

With provisioned Timestream Compute Units (TCUs), you can allocate a fixed number of TCUs to your account, ensuring predictable performance and cost for your queries. By provisioning TCUs, you gain greater control over compute capacity, enabling you to optimize both performance and query costs based on your application's specific needs.

Topics

- Benefits of Provisioning TCU
- How Provisioned TCU Works
- Monitoring Provisioned TCU usage
- Modifying your Provisioned TCUs
- Pricing for Provisioned TCUs

Benefits of Provisioning TCU

Provisioning TCU provides several benefits for customers with dedicated workloads, including:

1. **Predictable Performance:** By allocating a fixed number of TCUs, you ensure consistent performance for your queries.

2. **Cost Control:** With provisioned TCU, you can better predict and manage your costs, as you are only charged for the duration of the provisioned TCUs.
3. **Flexibility:** Provisioned TCU ensures that your workload has dedicated compute resources and you can adjust the number of provisioned TCUs to match your workload requirements, providing the required scalability as your application's needs change.

How Provisioned TCU Works

Each Timestream Compute Unit (TCU) is comprised of 4 vCPUs and 16GB of memory. To provision TCUs, use the AWS Management Console or the UpdateAccountSettings API operation to allocate a fixed number of TCUs to your account, which are then dedicated to your workload. This ensures predictable performance and cost for your queries. The minimum number of provisioned TCUs is 4, with subsequent increments also in multiples of 4 (e.g., 4, 8, 12, 16). Once provisioned, you can run your query workloads uninterrupted. As your workload demands change, you can adjust the provisioned TCUs using the AWS Management Console or the UpdateAccountSettings API operation at any time. However, you can only decrease the number of TCUs after a minimum of 1 hour has passed since provisioning them.

For example, if you provision 8 TCUs at 10:00 AM, you will be charged for a minimum of 1 hour, until 11:00 AM. During this time, you can increment the TCUs to 12 or more, but you cannot decrement them until 11:00 AM.

The time it takes to provision the requested Timestream Compute Units (TCUs) in your account varies depending on the number of TCUs requested. For example, provisioning 100 TCUs could take up to 30 minutes. However, you will only be charged for the resources once they are provisioned and available to serve your query workload. To ensure a smooth experience during planned increases in usage, we recommend provisioning the required resources in advance. This allows sufficient time for the resources to become available and ensures that your workload can be handled without interruption.

Monitoring Provisioned TCU usage

To monitor your provisioned TCU usage, you can use the following CloudWatch metrics:

- **Provisioned QueryTCU:** This metric specifies the number of TCUs provisioned in your account.
- **QueryTCU:** This metric specifies the number of TCUs used by your workload.
- **InsufficientTCUThrottles:** This metric specifies the number of queries throttled due to insufficient compute capacity.

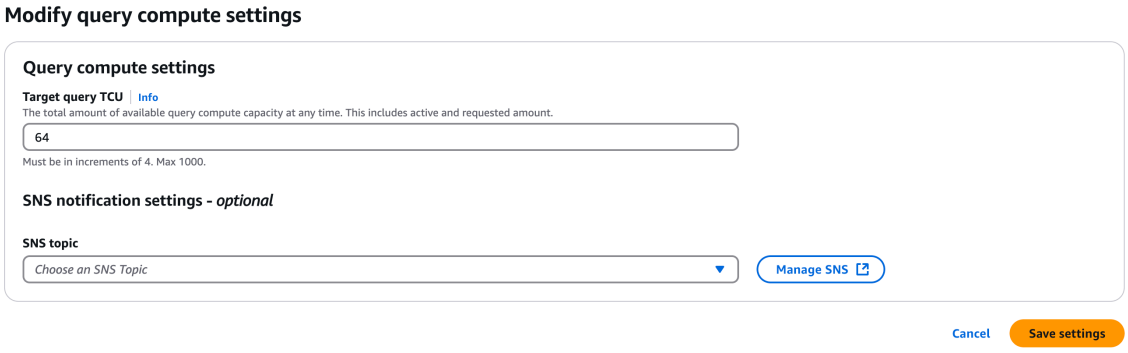
Modifying your Provisioned TCUs

You can adjust the number of provisioned Timestream Compute Units (TCUs) to match your changing workload demands using the AWS Management Console, AWS Command Line Interface (CLI), or AWS SDKs.

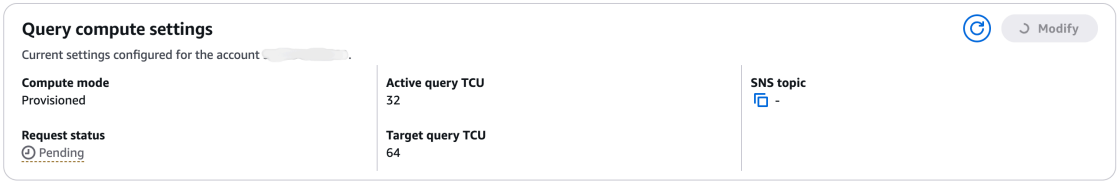
To view the current number of provisioned TCUs in your account, navigate to the "Admin Dashboard" section in the AWS Management Console. From there, you can easily monitor and manage your provisioned TCUs.

In the Query Compute Settings, you can verify that the compute mode is set to "Provisioned" and view the current number of provisioned Timestream Compute Units (TCUs) in your account, which is displayed as "Active Query TCU". The default value is 0. You need to provision TCUs before you run your query workload.

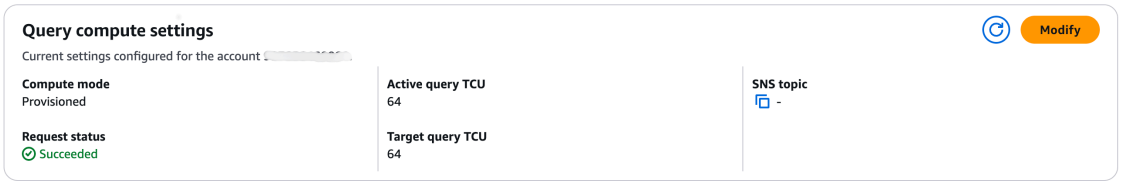
To modify the query compute settings, click the "Modify" button. For instance, if you want to increase the provisioned TCUs from 32 to 64, simply enter your desired target value (64) in the "Target Query TCU" field. Additionally, you can specify an Amazon Simple Notification Service (SNS) topic to receive a notification when the provisioning process is complete.



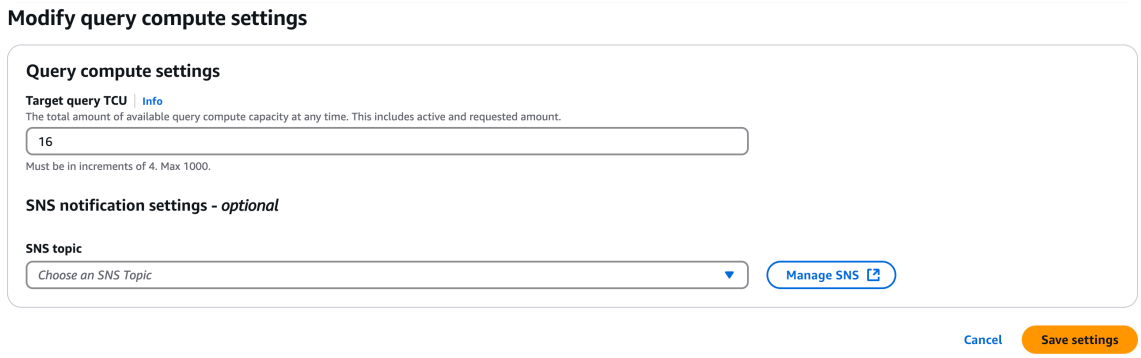
After confirming your desired configuration by selecting "Save settings", you will see that the current request status is updated to "Pending". The "Target Query TCU" field will now reflect the desired number of compute units, which is 64 in this case, indicating that the provisioning process has been initiated and is awaiting completion.



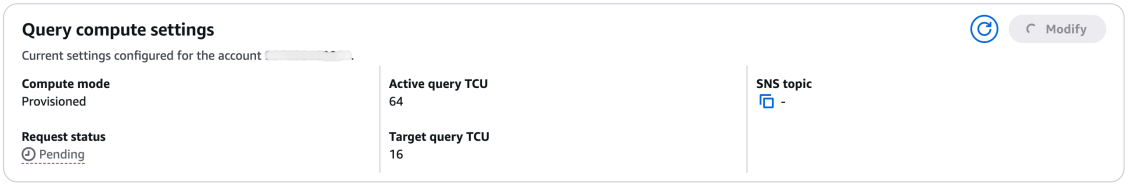
Once provisioned, the "Active Query TCU" field will be updated to reflect the new provisioned capacity of 64 Timestream Compute Units, indicating that the provisioning process is complete and the additional resources are now available for use in your account.



To reduce the number of provisioned Timestream Compute Units (TCUs) in your account, follow the same steps as before and enter your desired target value. For example, if you want to decrease the provisioned TCUs to 16, simply set the "Target Query TCU" field to 16. Please note that you can only decrease the number of provisioned TCUs after a minimum of 1 hour has passed since the last provisioning request. This means that if you provisioned or modified your TCUs within the last hour, you will need to wait until the 1-hour window has elapsed before you can decrement the TCUs.



After requesting a decrease in provisioned Timestream Compute Units (TCUs), the service will decrement the TCUs when it determines it is safe to do so, which may take up to a few minutes. During this time, the "Target Query TCU" field will continue to display the desired target value, in this case, 16 TCUs, indicating the pending change. Once the decrement is complete, the "Active Query TCU" field will be updated to reflect the new provisioned capacity of 16 TCUs.



Once the request is successfully completed, the "Active Query TCU" field will be updated to reflect the new provisioned capacity of 16 Timestream Compute Units (TCUs). If you no longer anticipate any query workload, you can further decrement the provisioned TCUs to 0, effectively releasing all provisioned resources and stopping any associated charges.

Pricing for Provisioned TCUs

You are charged for the duration of the Timestream Compute Units (TCUs) provisioned in your account, with a minimum charge of 1 hour. After the first hour, the TCUs are metered per second.

To calculate the total metered hours, multiply the number of provisioned TCUs by the duration of use. For example: If you provision 16 TCUs for 2 hours, the total metered hours are 16 TCU * 2 hours = 32 TCU-hours. If you provision 16 TCUs for 4 hours, then decrement to 8 TCUs and use them for 6 hours, the total metered hours are 16 TCU * 4 hours + 8 TCU * 6 hours = 112 TCU-hours.

Your total spend will depend on the prevailing TCU-hour cost in your region. Please refer to the Amazon Timestream Pricing page for detailed information.

Best Practices for managing Provisioned TCU

To get the most out of the Provisioned TCU feature, follow these best practices:

- **Monitor your workload:** Monitor your workload's performance, QueryTCU used and view `InsufficientTCUThrottles` to understand your usage patterns and adjust your provisioned TCUs accordingly.
- **Pro-active adjustment:** Increase or decrease provisioned TCUs based on observed trends and anticipated workload changes. Make adjustments for your peak and off-peak periods.
- **Maintain Headroom:** Maintain your consumed QueryTCU to within 80% - 90% of your ProvisionedQueryTCU to handle unexpected spikes.
- **Optimize Queries:** Leverage features such as Query Insights and follow Timestream Query best practices to optimize queries for reduced compute usage.
- **Implement Retries:**Timestream for LiveAnalytics Query SDK supports a retry mechanism with a default of 3 retries. Adjust the value accordingly to handle occasional and unanticipated bursts.

MaxQuery TCU

This setting specifies the maximum number of compute units the service will use at any point in time to serve your queries. To run queries, you must set the minimum capacity to 4 TCUs. You can set the maximum number of TCUs in multiples of 4, for example, 4, 8, 16, 32, and so on. You're charged only for the compute resources you use for your workload. For example, if you set the maximum TCUs to 128, but consistently use only 8 TCUs. You'll be charged only for the duration during which you used the 8 TCUs. The default `MaxQueryTCU` in your account is set

to 200. You can adjust `MaxQueryTCU` from 4 to 1000, using the AWS Management Console or UpdateAccountSettings API operation with the AWS SDK or AWS CLI.

We recommend setting the `MaxQueryTCU` for your account. Setting a maximum TCU limit helps control costs by restricting the number of compute units the service can use for your query workload. This allows you to better predict and manage your query spending.

Billing for TCU

Each TCU is billed on an hourly basis with per-second granularity and for a minimum of 30 seconds. The usage unit of these compute units is TCU-hour.

When you run queries, you're billed for the TCUs used during the query execution time, measured in TCU-hours. For example:

- Your workload uses 20 TCUs for 3 hours. You're billed for 60 TCU-hours (20 TCUs x 3 hours).
- Your workload uses 10 TCUs for 30 minutes, and then 20 TCUs for the next 30 minutes. You're billed for 15 TCU-hours (10 TCUs x 0.5 hours + 20 TCUs x 0.5 hours).

The pricing per TCU-hour varies by AWS Region. Refer to Amazon Timestream pricing for additional details. As your workload grows, the service automatically scales the compute capacity up to the specified maximum TCU limit (`MaxQueryTCU`) to maintain consistent performance. The `MaxQueryTCU` setting acts as a ceiling for the compute capacity that the service can scale to. This setting helps you to control the number of compute resources and as a result their cost.

Configuring TCU

When you onboard the service, each AWS account has a default `MaxQueryTCU` limit of 200. You can update this limit as required at any point in time using the AWS Management Console or UpdateAccountSettings API operation with the AWS SDK or AWS CLI.

If you're unsure about the values to configure, monitor the `QueryTCU` metric for your account. This metric is available in the AWS Management Console and Amazon CloudWatch. This metric provides insight into the maximum number of TCUs used at a minute granularity. Based on historical data and your estimation of future growth, set the `MaxQueryTCU` to accommodate the spikes in your usage. We recommend having a headroom of at least 4-16 TCUs above your peak usage. For example, if your peak `QueryTCU` in the last 30 days was 128, we recommend setting `MaxQueryTCU` between 132 to 144.

Estimating required compute units

Compute units can process queries concurrently. To determine the number of compute units required, consider the general guidelines in the following table:

Concurrent queries	TCUs
7	4
14	8
21	12

Note

- These are general guidelines and the actual number of compute units required depends on several factors, such as:
  - The effective concurrency of queries.
  - Query patterns.
  - The number of partitions scanned.
  - Other workload-specific characteristics.
- This guideline pertains to queries that scan for the last few minutes to an hour of data and adhere to the Timestream query best practices and Data modeling guidelines.
- Monitor your application's performance and the `QueryTCU` metric to adjust the compute units, as required.

When to increase MaxQueryTCU

You should consider increasing the `MaxQueryTCU` in the following scenarios:

- Your peak query consumption is approaching or reaching the current configured maximum query TCU. We recommend setting the maximum query TCU at least 4-16 TCUs higher than your peak consumption.

- Your queries are returning a 4xx error with the message MaxQueryTCU exceeded. If you anticipate a planned increase in your workload, revisit and adjust the configured maximum query TCU accordingly.

When to decrease MaxQueryTCU

You should consider decreasing the `MaxQueryTCU` in the following scenarios:

- Your workload has a predictable and stable usage pattern, and you have a good understanding of your compute usage requirements. Lowering the maximum query TCU to within 4-16 TCU above your peak consumption can help prevent unintentional usage and costs. You can modify the value using the UpdateAccountSettings API operation.
- Your workload's peak usage has decreased over time, either due to changes in your application or user behavior patterns. Lowering the maximum TCU can help mitigate unintentional costs.

> **Note**
>
> Depending on your current usage, reducing the maximum TCU limit change might take up to 24 hours to be effective. You're billed only for the TCUs that your queries actually consume. Having a higher maximum query TCU limit does not impact your costs unless those TCUs are used by your workload.

Monitoring usage with CloudWatch metrics

To monitor your TCU usage, Timestream for Live Analytics provides the following CloudWatch metric: `QueryTCU`. This metric specifies the number of compute units used in a minute and is emitted every minute. You can choose to monitor the maximum and minimum TCUs used in a minute. You can also set alarms on this metric to track your query costs in real-time.

Understanding variations in compute units usage

The number of compute resources required for your queries can either increase or decrease based on several parameters. For example, data volume, data ingestion patterns, query latency, query shape, query efficiency, and query combinations that use real-time and analytical queries. These parameters can lead to either higher or lower TCU units required for your workload. In a steady

state where these parameters don't change, you might observe that the number of compute units required for your workload decrease. Consequently, this can lower your monthly cost.

Additionally, if any of these parameters in your workload or data change, the number of compute units required might increase. When Timestream receives a query, depending upon the data partitions the query accesses, Timestream decides the number of compute resources to performantly address the query.

At periodic intervals, based on your ingest and query access patterns, Timestream optimizes the data layout. Timestream performs the optimization by clubbing less accessed partitions into a single partition or splitting a hot partition into multiple partitions for performance. Consequently, the compute capacity used by the same query might vary slightly at different points in time.

> **Opting-in to use TCU pricing for your queries**
>
> As an existing user, you can do a one-time opt-in to use TCUs for better cost management and removal of per query minimum bytes metered. You can opt-in using the AWS Management Console or UpdateAccountSettings API operation with the AWS SDK or AWS CLI. In the API operation, set the `QueryPricingModel` parameter to `COMPUTE_UNITS`. Opting into the compute-based pricing model is an irreversible change.

Accessing Timestream for LiveAnalytics

You can access Timestream for LiveAnalytics using the console, CLI or the API. For information about accessing Timestream for LiveAnalytics, review the following:

Topics

- Sign up for an AWS account
- Provide Timestream for LiveAnalytics access
- Using the console
- Accessing Amazon Timestream for LiveAnalytics using the AWS CLI
- Using the API
- Using the AWS SDKs

Sign up for an AWS account

To get started with AWS, you need an AWS account. For information about creating an AWS account, see Getting started with an AWS account in the *AWS Account Management Reference Guide*.

Provide Timestream for LiveAnalytics access

The permissions that are required to access Timestream for LiveAnalytics are already granted to the administrator. For other users, you should grant them Timestream for LiveAnalytics access using the following policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "timestream:*",
        "kms:DescribeKey",
        "kms:CreateGrant",
        "kms:Decrypt",
        "dbqms:CreateFavoriteQuery",
        "dbqms:DescribeFavoriteQueries",
        "dbqms:UpdateFavoriteQuery",
        "dbqms:DeleteFavoriteQueries",
        "dbqms:GetQueryString",
        "dbqms:CreateQueryHistory",
        "dbqms:UpdateQueryHistory",
        "dbqms:DeleteQueryHistory",
        "dbqms:DescribeQueryHistory",
        "s3:ListAllMyBuckets"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> For information about dbqms, see Actions, resources, and condition keys for Database Query Metadata Service. For information about kms see Actions, resources, and condition keys for AWS Key Management Service.

Using the console

You can use the AWS Management Console for Timestream Live Analytics to create, edit, delete, describe, and list databases and tables. You can also use the console to run queries.

Topics

- Tutorial
- Create a database
- Create a table
- Run a query
- Create a scheduled query
- Delete a scheduled query
- Delete a table
- Delete a database
- Edit a table
- Edit a database

Tutorial

This tutorial shows you how to create a database populated with sample data sets and run sample queries. The sample datasets used in this tutorial are frequently seen in IoT and DevOps scenarios. The IoT dataset contains time series data such as the speed, location, and load of a truck, to streamline fleet management and identify optimization opportunities. The DevOps dataset contains EC2 instance metrics such as CPU, network, and memory utilization to improve application performance and availability. Here's a video tutorial for the instructions described in this section

Follow these steps to create a database populated with the sample data sets and run sample queries using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Databases**
3. Click on **Create database**.
4. On the create database page, enter the following:
   - **Choose configuration**—Select **Sample database**.
   - **Name**—Enter a database name of your choice.
   - **Choose sample datasets**—Select **IoT** and **DevOps**.
   - Click on **Create database** to create a database containing two tables—IoT and DevOps populated with sample data.
5. In the navigation pane, choose **Query editor**
6. Select **Sample queries** from the top menu.
7. Click on one of the sample queries. This will take you back to the query editor with the editor populated with the sample query.
8. Click **Run** to run the query and see query results.

Create a database

Follow these steps to create a database using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Databases**
3. Click on **Create database**.
4. On the create database page, enter the following.
   - **Choose configuration**—Select **Standard database**.
   - **Name**—Enter a database name of your choice.
   - **Encryption** —Choose a KMS key or use the default option, where Timestream Live Analytics will create a KMS key in your account if one does not already exist.
5. Click on **Create database** to create a database.

Create a table

Follow these steps to create a table using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Tables**
3. Click on **Create table**.
4. On the create table page, enter the following.
   - **Database name**—Select the name of the database created in Create a database.
   - **Table name**—Enter a table name of your choice.
   - **Memory store retention**—Specify how long you want to retain data in the memory store. The memory store processes incoming data, including late arriving data (data with a timestamp earlier than the current time) and is optimized for fast point-in-time queries.
   - **Magnetic store retention**—Specify how long you want to retain data in the magnetic store. The magnetic store is meant for long term storage and is optimized for fast analytical queries.
5. Click on **Create table**.

Run a query

Follow these steps to run queries using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Query editor**
3. In the left pane, select the database created in Create a database.
4. In the left pane, select the database created in Create a table.
5. In the query editor, you can run a query. To see the latest 10 rows in the table, run:

```
SELECT * FROM <database_name>.<table_name> ORDER BY time DESC LIMIT 10
```

6. (Optional) Turn on **Enable Insights** to get insights about the efficiency of your queries.

Create a scheduled query

Follow these steps to create a scheduled query using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Scheduled queries**.

3. Click on **Create scheduled query**.
4. In the **Query Name** and **Destination Table** sections, enter the following.
   - **Name**—Enter a query name.
   - **Database name**—Select the name of the database created in Create a database.
   - **Table name**—Select the name of the table created in Create a table.
5. In the **Query Statement** section, enter a valid query statement. Then click **Validate query**.
6. From **Destination table model**, define the model for any undefined attributes. You can use **Visual builder** or JSON.
7. In the **Run schedule** section, choose **Fixed rate** or **Chron expression**.For chron expressions, refer to Schedule Expressions for Scheduled Queries for more details on schedule expressions.
8. In the **SNS topic** section, enter the SNS topic that will be used to for notification.
9. In the **Error log report** section enter the S3 location that will be used to report errors.

   Choose the **Encryption key type**.
10. In the **Security settings** section from **AWS KMS key**, choose the type of AWS KMS key.

    Enter the **IAM role** that Timestream for LiveAnalytics will use to run the scheduled query. Refer to the IAM policy examples for scheduled queries for details on the required permissions and trust relationship for the role.
11. Click **Create scheduled query**.

Delete a scheduled query

Follow these steps to delete or disable a scheduled query using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Scheduled queries**
3. Select the scheduled query created in Create a scheduled query.
4. Select **Actions**.
5. Choose **Disable** or **Delete**.
6. If you selected Delete, confirm the action and select **Delete**.

Delete a table

Follow these steps to delete a database using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Tables**
3. Select the table that you created in Create a table.
4. Click **Delete**.
5. Type *delete* in the confirmation box.

Delete a database

Follow these steps to delete a database using the AWS Console:

1. Open the AWS Console.
2. In the navigation pane, choose **Databases**
3. Select the database that you created in **Create a database**.
4. Click **Delete**.
5. Type *delete* in the confirmation box.

Edit a table

Follow these steps to edit a table using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Tables**
3. Select the table that you created in Create a table.
4. Click **Edit**
5. Edit the table details and save.

   - **Memory store retention**—Specify how long you want to retain data in the memory store. The memory store processes incoming data, including late arriving data (data with a timestamp earlier than the current time) and is optimized for fast point-in-time queries.

- **Magnetic store retention**—Specify how long you want to retain data in the magnetic store. The magnetic store is meant for long term storage and is optimized for fast analytical queries.

Edit a database

Follow these steps to edit a database using the AWS Console.

1. Open the AWS Console.
2. In the navigation pane, choose **Databases**
3. Select the database that you created in **Create a database**.
4. Click **Edit**
5. Edit the database details and save.

Accessing Amazon Timestream for LiveAnalytics using the AWS CLI

You can use the AWS Command Line Interface (AWS CLI) to control multiple AWS services from the command line and automate them through scripts. You can use the AWS CLI for ad hoc operations. You can also use it to embed Amazon Timestream for LiveAnalytics operations within utility scripts.

Before you can use the AWS CLI with Timestream for LiveAnalytics, you must get credentials that grant programmatic access. For more information, see Provide Timestream for LiveAnalytics access.

For a complete listing of all the commands available for the Timestream for LiveAnalytics Query API in the AWS CLI, see the AWS CLI Command Reference.

For a complete listing of all the commands available for the Timestream for LiveAnalytics Write API in the AWS CLI, see the AWS CLI Command Reference.

Topics

- Downloading and configuring the AWS CLI
- Using the AWS CLI with Timestream for LiveAnalytics

Downloading and configuring the AWS CLI

The AWS CLI runs on Windows, macOS, or Linux. To download, install, and configure it, follow these steps:

1. Download the AWS CLI at http://aws.amazon.com/cli.
2. Follow the instructions for Installing the AWS CLI and Configuring the AWS CLI in the *AWS Command Line Interface User Guide*.

Using the AWS CLI with Timestream for LiveAnalytics

The command line format consists of an Amazon Timestream for LiveAnalytics operation name, followed by the parameters for that operation. The AWS CLI supports a shorthand syntax for the parameter values, in addition to JSON.

Use `help` to list all available commands in Timestream for LiveAnalytics. For example:

```
aws timestream-write help
```

```
aws timestream-query help
```

You can also use `help` to describe a specific command and learn more about its usage:

```
aws timestream-write create-database help
```

For example, to create a database:

```
aws timestream-write create-database --database-name myFirstDatabase
```

To create a table with magnetic store writes enabled:

```
aws timestream-write create-table \
--database-name metricsdb \
--table-name metrics \
--magnetic-store-write-properties "{\"EnableMagneticStoreWrites\": true}"
```

To write data using single-measure records:

```
aws timestream-write write-records \
--database-name metricsdb \
--table-name metrics \
--common-attributes "{\"Dimensions\":[{\"Name\":\"asset_id\", \"Value\":\"100\"}],
 \"Time\":\"1631051324000\",\"TimeUnit\":\"MILLISECONDS\"}" \
```

```
--records "[{\"MeasureName\":\"temperature\", \"MeasureValueType\":\"DOUBLE\",
\"MeasureValue\":\"30\"},{\"MeasureName\":\"windspeed\", \"MeasureValueType\":\"DOUBLE
\",\"MeasureValue\":\"7\"},{\"MeasureName\":\"humidity\", \"MeasureValueType\":\"DOUBLE
\",\"MeasureValue\":\"15\"},{\"MeasureName\":\"brightness\", \"MeasureValueType\":
\"DOUBLE\",\"MeasureValue\":\"17\"}]"
```

To write data using multi-measure records:

```
# wide model helper method to create Multi-measure records
function ingest_multi_measure_records {
  epoch=`date +%s`
  epoch+=$i

  # multi-measure records
  aws timestream-write write-records \
  --database-name $src_db_wide \
  --table-name $src_tbl_wide \
  --common-attributes "{\"Dimensions\":[{\"Name\":\"device_id\", \
              \"Value\":\"12345678\"},\
            {\"Name\":\"device_type\", \"Value\":\"iPhone\"}, \
            {\"Name\":\"os_version\", \"Value\":\"14.8\"}, \
            {\"Name\":\"region\", \"Value\":\"us-east-1\"} ], \
            \"Time\":\"$epoch\",\"TimeUnit\":\"MILLISECONDS\"}" \
--records "[{\"MeasureName\":\"video_metrics\", \"MeasureValueType\":\"MULTI\", \
  \"MeasureValues\": \
  [{\"Name\":\"video_startup_time\",\"Value\":\"0\",\"Type\":\"BIGINT\"}, \
  {\"Name\":\"rebuffering_ratio\",\"Value\":\"0.5\",\"Type\":\"DOUBLE\"}, \
  {\"Name\":\"video_playback_failures\",\"Value\":\"0\",\"Type\":\"BIGINT\"}, \
  {\"Name\":\"average_frame_rate\",\"Value\":\"0.5\",\"Type\":\"DOUBLE\"}]}]" \
--endpoint-url $ingest_endpoint \
  --region  $region
}

# create 5 records
for i in {100..105};
  do ingest_multi_measure_records $i;
done
```

To query a table:

```
aws timestream-query query \
--query-string "SELECT time, device_id, device_type, os_version,
region, video_startup_time, rebuffering_ratio, video_playback_failures, \
```

```
average_frame_rate \
FROM metricsdb.metrics \
where time >= ago (15m)"
```

To create a scheduled query:

```
aws timestream-query create-scheduled-query \
  --name scheduled_query_name \
  --query-string "select bin(time, 1m) as time, \
          avg(measure_value::double) as avg_cpu, min(measure_value::double) as min_cpu,
 region \
          from $src_db.$src_tbl where measure_name = 'cpu' \
          and time BETWEEN @scheduled_runtime - (interval '5' minute)  AND
 @scheduled_runtime \
          group by region, bin(time, 1m)" \
  --schedule-configuration "{\"ScheduleExpression\":\"$cron_exp\"}" \
  --notification-configuration "{\"SnsConfiguration\":{\"TopicArn\":\"$sns_topic_arn
\"}}" \
  --scheduled-query-execution-role-arn "arn:aws:iam::452360119086:role/
TimestreamSQExecutionRole" \
  --target-configuration "{\"TimestreamConfiguration\":{\
          \"DatabaseName\": \"$dest_db\",\
          \"TableName\": \"$dest_tbl\",\
          \"TimeColumn\":\"time\",\
          \"DimensionMappings\":[{\
            \"Name\": \"region\", \"DimensionValueType\": \"VARCHAR\"
          }],\
          \"MultiMeasureMappings\":{\
            \"TargetMultiMeasureName\": \"mma_name\",
            \"MultiMeasureAttributeMappings\":[{\
              \"SourceColumn\": \"avg_cpu\", \"MeasureValueType\": \"DOUBLE\",
 \"TargetMultiMeasureAttributeName\": \"target_avg_cpu\"
            },\
            { \
              \"SourceColumn\": \"min_cpu\", \"MeasureValueType\": \"DOUBLE\",
 \"TargetMultiMeasureAttributeName\": \"target_min_cpu\"
            }] \
          }\
          }}" \
  --error-report-configuration "{\"S3Configuration\": {\
        \"BucketName\": \"$s3_err_bucket\",\
        \"ObjectKeyPrefix\": \"scherrors\",\
        \"EncryptionOption\": \"SSE_S3\"\
```

```
        }\
      }"
```

Using the API

In addition to the SDKs, Amazon Timestream for LiveAnalytics provides direct REST API access via the *endpoint discovery pattern*. The endpoint discovery pattern is described below, along with its use cases.

The endpoint discovery pattern

Because Timestream Live Analytics's SDKs are designed to transparently work with the service's architecture, including the management and mapping of the service endpoints, it is recommended that you use the SDKs for most applications. However, there are a few instances where use of the Timestream for LiveAnalytics REST API endpoint discovery pattern is necessary:

- You are using VPC endpoints (AWS PrivateLink) with Timestream for LiveAnalytics
- Your application uses a programming language that does not yet have SDK support
- You require better control over the client-side implementation

This section includes information on how the endpoint discovery pattern works, how to implement the endpoint discovery pattern, and usage notes. Select a topic below to learn more.

Topics

- How the endpoint discovery pattern works
- Implementing the endpoint discovery pattern

How the endpoint discovery pattern works

Timestream is built using a cellular architecture to ensure better scaling and traffic isolation properties. Because each customer account is mapped to a specific cell in a region, your application must use the correct cell-specific endpoints that your account has been mapped to. When using the SDKs, this mapping is transparently handled for you and you do not need to manage the cell-specific endpoints. However, when directly accessing the REST API, you will need to manage and map the correct endpoints yourself. This process, the *endpoint discovery pattern*, is described below:

1. The endpoint discovery pattern starts with a call to the `DescribeEndpoints` action (described in the `DescribeEndpoints` section).

2. The endpoint should be cached and reused for the amount of time specified by the returned time-to-live (TTL) value (the `CachePeriodInMinutes`). Calls to the Timestream Live Analytics API can then be made for the duration of the TTL.
3. After the TTL expires, a new call to DescribeEndpoints should be made to refresh the endpoint (in other words, start over at Step 1).

> **Note**
>
> Syntax, parameters and other usage information for the `DescribeEndpoints` action are described in the API Reference. Note that the `DescribeEndpoints` action is available via both SDKs, and is identical for each.

For implementation of the endpoint discovery pattern, see Implementing the endpoint discovery pattern.

Implementing the endpoint discovery pattern

To implement the endpoint discovery pattern, choose an API (Write or Query), create a **DescribeEndpoints** request, and use the returned endpoint(s) for the duration of the returned TTL value(s). The implementation procedure is described below.

> **Note**
>
> Ensure you are familiar with the usage notes.

Implementation procedure

1. Acquire the endpoint for the API you would like to make calls against (Write or Query). using the `DescribeEndpoints` request.
   a. Create a request for `DescribeEndpoints` that corresponds to the API of interest (Write or Query) using one of the two endpoints described below. There are no input parameters for the request. Ensure that you read the notes below.

Write SDK:

```
ingest.timestream.<region>.amazonaws.com
```

Query SDK:

```
query.timestream.<region>.amazonaws.com
```

An example CLI call for region `us-east-1` follows.

```
REGION_ENDPOINT="https://query.timestream.us-east-1.amazonaws.com"
REGION=us-east-1
aws timestream-write describe-endpoints \
--endpoint-url $REGION_ENDPOINT \
--region $REGION
```

> **Note**
>
> The HTTP "Host" header *must* also contain the API endpoint. The request will fail if the header is not populated. This is a standard requirement for all HTTP/1.1 requests. If you use an HTTP library supporting 1.1 or later, the HTTP library should automatically populate the header for you.

> **Note**
>
> Substitute *<region>* with the region identifier for the region the request is being made in, e.g. `us-east-1`

b. Parse the response to extract the endpoint(s), and cache TTL value(s). The response is an array of one or more `Endpoint objects`. Each `Endpoint` object contains an endpoint address (`Address`) and the TTL for that endpoint (`CachePeriodInMinutes`).

2. Cache the endpoint for up to the specified TTL.
3. When the TTL expires, retrieve a new endpoint by starting over at step 1 of the Implementation.

Usage notes for the endpoint discovery pattern

- The **DescribeEndpoints** action is the only action that Timestream Live Analytics regional endpoints recognize.
- The response contains a list of endpoints to make Timestream Live Analytics API calls against.
- On successful response, there should be at least one endpoint in the list. If there is more than one endpoint in the list, any of them are equally usable for the API calls, and the caller may choose the endpoint to use at random.
- In addition to the DNS address of the endpoint, each endpoint in the list will specify a time to live (TTL) that is allowable for using the endpoint specified in minutes.
- The endpoint should be cached and reused for the amount of time specified by the returned TTL value (in minutes). After the TTL expires a new call to **DescribeEndpoints** should be made to refresh the endpoint to use, as the endpoint will no longer work after the TTL has expired.

Using the AWS SDKs

You can access Amazon Timestream using the AWS SDKs. Timestream supports two SDKs per language; namely, the Write SDK and the Query SDK. The Write SDK is used to perform CRUD operations and to insert your time series data into Timestream. The Query SDK is used to query your existing time series data stored in Timestream.

Once you've completed the necessary prerequisites for your SDK of choice, you can get started with the Code samples.

Topics

- Java
- Java v2
- Go
- Python
- Node.js
- .NET

Java

To get started with the Java 1.0 SDK and Amazon Timestream, complete the prerequisites, described below.

Once you've completed the necessary prerequisites for the Java SDK, you can get started with the Code samples.

Prerequisites

Before you get started with Java, you must do the following:

1. Follow the AWS setup instructions in Accessing Timestream for LiveAnalytics.
2. Set up a Java development environment by downloading and installing the following:
   - Java SE Development Kit 8 (such as Amazon Corretto 8).
   - Java IDE (such as Eclipse or IntelliJ).

     For more information, see Getting Started with the AWS SDK for Java
3. Configure your AWS credentials and Region for development:
   - Set up your AWS security credentials for use with the AWS SDK for Java.
   - Set your AWS Region to determine your default Timestream for LiveAnalytics endpoint.

Using Apache Maven

You can use Apache Maven to configure and build AWS SDK for Java projects.

> **Note**
>
> To use Apache Maven, ensure your Java SDK and runtime are 1.8 or higher.

You can configure the AWS SDK as a Maven dependency as described in Using the SDK with Apache Maven.

You can run compile and run your source code with the following command:

```
mvn clean compile
```

```
mvn exec:java -Dexec.mainClass=<your source code Main class>
```

> **Note**
>
> `<your source code Main class>` is the path to your Java source code's main class.

Setting your AWS credentials

The AWS SDK for Java requires that you provide AWS credentials to your application at runtime. The code examples in this guide assume that you are using an AWS credentials file, as described in Set up AWS Credentials and Region for Development in the *AWS SDK for Java Developer Guide*.

The following is an example of an AWS credentials file named `~/.aws/credentials`, where the tilde character (~) represents your home directory.

```
[default]
aws_access_key_id = AWS access key ID goes here
aws_secret_access_key = Secret key goes here
```

Java v2

To get started with the Java 2.0 SDK and Amazon Timestream, complete the prerequisites, described below.

Once you've completed the necessary prerequisites for the Java 2.0 SDK, you can get started with the Code samples.

Prerequisites

Before you get started with Java, you must do the following:

1. Follow the AWS setup instructions in Accessing Timestream for LiveAnalytics.
2. You can configure the AWS SDK as a Maven dependency as described in Using the SDK with Apache Maven.
3. Set up a Java development environment by downloading and installing the following:
   - Java SE Development Kit 8 (such as Amazon Corretto 8).
   - Java IDE (such as Eclipse or IntelliJ).

For more information, see Getting Started with the AWS SDK for Java

Using Apache Maven

You can use Apache Maven to configure and build AWS SDK for Java projects.

> **Note**
>
> To use Apache Maven, ensure your Java SDK and runtime are 1.8 or higher.

You can configure the AWS SDK as a Maven dependency as described in Using the SDK with Apache Maven. The changes required to the pom.xml file are described here.

You can run compile and run your source code with the following command:

```
mvn clean compile
mvn exec:java -Dexec.mainClass=<your source code Main class>
```

> **Note**
>
> `<your source code Main class>` is the path to your Java source code's main class.

Go

To get started with the Go SDK and Amazon Timestream, complete the prerequisites, described below.

Once you've completed the necessary prerequisites for the Go SDK, you can get started with the Code samples.

Prerequisites

1. Download the GO SDK 1.14.
2. Configure the GO SDK.
3. Construct your client.

Python

To get started with the Python SDK and Amazon Timestream, complete the prerequisites, described below.

Once you've completed the necessary prerequisites for the Python SDK, you can get started with the Code samples.

Prerequisites

To use Python, install and configure Boto3, following the instructions here.

Node.js

To get started with the Node.js SDK and Amazon Timestream, complete the prerequisites, described below.

Once you've completed the necessary prerequisites for the Node.js SDK, you can get started with the Code samples.

Prerequisites

Before you get started with Node.js, you must do the following:

1. Install Node.js.
2. Install the AWS SDK for JavaScript.

.NET

To get started with the .NET SDK and Amazon Timestream, complete the prerequisites, described below.

Once you've completed the necessary prerequisites for the .NET SDK, you can get started with the Code samples.

Prerequisites

Before you get started with .NET, install the required NuGet packages and ensure that AWSSDK.Core version is 3.3.107 or newer by running the following commands:

```
dotnet add package AWSSDK.Core
```

```
dotnet add package AWSSDK.TimestreamWrite
dotnet add package AWSSDK.TimestreamQuery
```

Getting started

This section includes a tutorial to get you started with Amazon Timestream Live Analytics, as well as instructions for setting up a fully functional sample application. You can get started with the tutorial or the sample application by selecting one of the links below.

Topics

- Tutorial
- Sample application

Tutorial

This tutorial shows you how to create a database populated with sample data sets and run sample queries. The sample data sets used in this tutorial are frequently seen in IoT and DevOps scenarios. The IoT data set contains time series data such as the speed, location, and load of a truck, to streamline fleet management and identify optimization opportunities. The DevOps data set contains EC2 instance metrics such as CPU, network, and memory utilization to improve application performance and availability. Here's a video tutorial for the instructions described in this section.

Follow these steps to create a database populated with the sample data sets and run sample queries using the AWS Console:

Using the console

Follow these steps to create a database populated with the sample data sets and run sample queries using the AWS Console:

1. Open the AWS Console.
2. In the navigation pane, choose **Databases**.
3. Click on **Create database**.
4. On the create database page, enter the following:
    - **Choose configuration**—Select **Sample database**.

- **Name**—Enter a database name of your choice.

> **Note**
>
> After creating a database with sample data sets, to use the sample queries which are available in the console, you can adjust the database name referenced in the query to match the database name you enter here. There are sample queries for each combination of sample data set and type of time series records.

- **Choose sample data sets**—Select **IoT** and **DevOps**.
- **Choose the type of time series records**—Select **Multi-measure records**.
- Click on **Create database** to create a database containing two tables populated with sample data. The table names for sample data sets with multi-measure records are `DevOpsMulti` and `IoTMulti`. The table names for sample datasets with single-measure records are `DevOps` and `IoT`.

5. In the navigation pane, choose **Query editor**.
6. Select **Sample queries** from the top menu.
7. Click on one of the sample queries for a data set you chose when creating the sample database. This will take you back to the query editor with the editor populated with the sample query.
8. Adjust the database name for the sample query.
9. Click **Run** to run the query and see query results.

Using the SDKs

Timestream Live Analytics provides a fully functional sample application that shows you how to create a database and table, populate the table with ~126K rows of sample data, and run sample queries. The sample application is available in GitHub for Java, Python, Node.js, Go, and .NET.

1. Clone the GitHub repository Timestream Live Analytics sample applications following the instructions from GitHub.
2. Configure the AWS SDK to connect to Amazon Timestream Live Analytics following the instructions described in Using the AWS SDKs.
3. Compile and run the sample application using the instructions below:

   - Instructions for the Java sample application.

- Instructions for the Java v2 sample application.
- Instructions for the Go sample application.
- Instructions for the Python sample application.
- Instructions for the Node.js sample application.
- Instructions for the .NET sample application.

Sample application

Timestream ships with a fully functional sample application that shows how to create a database and table, populate the table with ~126K rows of sample data, and run sample queries. Follow the steps below to get started with the sample application in any of the supported languages:

Java

1. Clone the GitHub repository Timestream for LiveAnalytics sample applications following the instructions from GitHub.
2. Configure the AWS SDK to connect to Timestream for LiveAnalytics following the instructions described in Getting Started with Java.
3. Run the Java sample application following the instructions described here

Java v2

1. Clone the GitHub repository Timestream for LiveAnalytics sample applications following the instructions from GitHub.
2. Configure the AWS SDK to connect to Amazon Timestream for LiveAnalytics following the instructions described in Getting Started with Java v2.
3. Run the Java 2.0 sample application following the instructions described here

Go

1. Clone the GitHub repository Timestream for LiveAnalytics sample applications following the instructions from GitHub.
2. Configure the AWS SDK to connect to Amazon Timestream for LiveAnalytics following the instructions described in Getting Started with Go.
3. Run the Go sample application following the instructions described here

Python

1. Clone the GitHub repository Timestream for LiveAnalytics sample applications following the instructions from GitHub.
2. Configure the AWS SDK to connect to Amazon Timestream for LiveAnalytics following the instructions described in Python.
3. Run the Python sample application following the instructions described here

Node.js

1. Clone the GitHub repository Timestream for LiveAnalytics sample applications following the instructions from GitHub.
2. Configure the AWS SDK to connect to Amazon Timestream for LiveAnalytics following the instructions described in Getting Started with Node.js.
3. Run the Node.js sample application following the instructions described here

.NET

1. Clone the GitHub repository Timestream for LiveAnalytics sample applications following the instructions from GitHub.
2. Configure the AWS SDK to connect to Amazon Timestream for LiveAnalytics following the instructions described in Getting Started with .NET.
3. Run the .NET sample application following the instructions described here

Code samples

You can access Amazon Timestream using the AWS SDKs. Timestream supports two SDKs per language; namely, the Write SDK and the Query SDK. The Write SDK is used to perform CRUD operations and to insert your time series data into Timestream. The Query SDK is used to query your existing time series data stored in Timestream. Select a topic from the list below for more details, including code samples for each of the supported SDKs.

Topics

- Write SDK client
- Query SDK client

- Create database
- Describe database
- Update database
- Delete database
- List databases
- Create table
- Describe table
- Update table
- Delete table
- List tables
- Write data (inserts and upserts)
- Run query
- Run UNLOAD query
- Cancel query
- Create batch load task
- Describe batch load task
- List batch load tasks
- Resume batch load task
- Create scheduled query
- List scheduled query
- Describe scheduled query
- Execute scheduled query
- Update scheduled query
- Delete scheduled query

Write SDK client

You can use the following code snippets to create a Timestream client for the Write SDK. The Write SDK is used to perform CRUD operations and to insert your time series data into Timestream.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
private static AmazonTimestreamWrite buildWriteClient() {
    final ClientConfiguration clientConfiguration = new ClientConfiguration()
            .withMaxConnections(5000)
            .withRequestTimeout(20 * 1000)
            .withMaxErrorRetry(10);

    return AmazonTimestreamWriteClientBuilder
            .standard()
            .withRegion("us-east-1")
            .withClientConfiguration(clientConfiguration)
            .build();
}
```

Java v2

```
private static TimestreamWriteClient buildWriteClient() {
    ApacheHttpClient.Builder httpClientBuilder =
            ApacheHttpClient.builder();
    httpClientBuilder.maxConnections(5000);

    RetryPolicy.Builder retryPolicy =
            RetryPolicy.builder();
    retryPolicy.numRetries(10);

    ClientOverrideConfiguration.Builder overrideConfig =
            ClientOverrideConfiguration.builder();
    overrideConfig.apiCallAttemptTimeout(Duration.ofSeconds(20));
    overrideConfig.retryPolicy(retryPolicy.build());

    return TimestreamWriteClient.builder()
            .httpClientBuilder(httpClientBuilder)
            .overrideConfiguration(overrideConfig.build())
            .region(Region.US_EAST_1)
            .build();
```

```
    }
```

Go

```
tr := &http.Transport{
        ResponseHeaderTimeout: 20 * time.Second,
        // Using DefaultTransport values for other parameters: https://golang.org/
pkg/net/http/#RoundTripper
        Proxy: http.ProxyFromEnvironment,
        DialContext: (&net.Dialer{
            KeepAlive: 30 * time.Second,
            DualStack: true,
            Timeout:   30 * time.Second,
        }).DialContext,
        MaxIdleConns:          100,
        IdleConnTimeout:       90 * time.Second,
        TLSHandshakeTimeout:   10 * time.Second,
        ExpectContinueTimeout: 1 * time.Second,
    }

    // So client makes HTTP/2 requests
    http2.ConfigureTransport(tr)

    sess, err := session.NewSession(&aws.Config{ Region: aws.String("us-east-1"),
 MaxRetries: aws.Int(10), HTTPClient: &http.Client{ Transport: tr }})
    writeSvc := timestreamwrite.New(sess)
```

Python

```
write_client = session.client('timestream-write', config=Config(read_timeout=20,
 max_pool_connections = 5000, retries={'max_attempts': 10}))
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

An additional command import is shown here. The `CreateDatabaseCommand` import is not required to create the client.

```
import { TimestreamWriteClient, CreateDatabaseCommand } from "@aws-sdk/client-
timestream-write";
```

```
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
var https = require('https');
var agent = new https.Agent({
    maxSockets: 5000
});
writeClient = new AWS.TimestreamWrite({
        maxRetries: 10,
        httpOptions: {
            timeout: 20000,
            agent: agent
        }
    });
```

.NET

```
var writeClientConfig = new AmazonTimestreamWriteConfig
{
    RegionEndpoint = RegionEndpoint.USEast1,
    Timeout = TimeSpan.FromSeconds(20),
    MaxErrorRetry = 10
};

var writeClient = new AmazonTimestreamWriteClient(writeClientConfig);
```

We recommend you use the following configuration.

- Set the SDK retry count to `10`.
- Use SDK `DEFAULT_BACKOFF_STRATEGY`.
- Set `RequestTimeout` to `20` seconds.
- Set the max connections to `5000` or higher.

Query SDK client

You can use the following code snippets to create a Timestream client for the Query SDK. The Query SDK is used to query your existing time series data stored in Timestream.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    private static AmazonTimestreamQuery buildQueryClient() {
        AmazonTimestreamQuery client =
AmazonTimestreamQueryClient.builder().withRegion("us-east-1").build();
        return client;
    }
```

Java v2

```
    private static TimestreamQueryClient buildQueryClient() {
        return TimestreamQueryClient.builder()
                .region(Region.US_EAST_1)
                .build();
    }
```

Go

```
sess, err := session.NewSession(&aws.Config{Region: aws.String("us-east-1")})
```

Python

```
query_client = session.client('timestream-query')
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Query Client - ,AWS SDK for JavaScript v3.

An additional command import is shown here. The `QueryCommand` import is not required to create the client.

```
import { TimestreamQueryClient, QueryCommand } from "@aws-sdk/client-timestream-
query";
```

```
const queryClient = new TimestreamQueryClient({ region: "us-east-1" });
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
queryClient = new AWS.TimestreamQuery();
```

.NET

```
var queryClientConfig = new AmazonTimestreamQueryConfig
{
    RegionEndpoint = RegionEndpoint.USEast1
};

var queryClient = new AmazonTimestreamQueryClient(queryClientConfig);
```

Create database

You can use the following code snippets to create a database.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
  public void createDatabase() {
       System.out.println("Creating database");
       CreateDatabaseRequest request = new CreateDatabaseRequest();
       request.setDatabaseName(DATABASE_NAME);
       try {
           amazonTimestreamWrite.createDatabase(request);
           System.out.println("Database [" + DATABASE_NAME + "] created
successfully");
       } catch (ConflictException e) {
           System.out.println("Database [" + DATABASE_NAME + "] exists. Skipping
database creation");
       }
```

```
    }
```

Java v2

```
    public void createDatabase() {
        System.out.println("Creating database");
        CreateDatabaseRequest request =
 CreateDatabaseRequest.builder().databaseName(DATABASE_NAME).build();
        try {
            timestreamWriteClient.createDatabase(request);
            System.out.println("Database [" + DATABASE_NAME + "] created
 successfully");
        } catch (ConflictException e) {
            System.out.println("Database [" + DATABASE_NAME + "] exists. Skipping
 database creation");
        }
    }
```

Go

```
// Create database.
    createDatabaseInput := &timestreamwrite.CreateDatabaseInput{
        DatabaseName: aws.String(*databaseName),
    }

    _, err = writeSvc.CreateDatabase(createDatabaseInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("Database successfully created")
    }

    fmt.Println("Describing the database, hit enter to continue")
```

Python

```
    def create_database(self):
        print("Creating Database")
        try:
            self.client.create_database(DatabaseName=Constant.DATABASE_NAME)
```

```
            print("Database [%s] created successfully." % Constant.DATABASE_NAME)
        except self.client.exceptions.ConflictException:
            print("Database [%s] exists. Skipping database creation" %
 Constant.DATABASE_NAME)
        except Exception as err:
            print("Create database failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class CreateDatabaseCommand and CreateDatabase.

```
import { TimestreamWriteClient, CreateDatabaseCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });

const params = {
    DatabaseName: "testDbFromNode"
};

const command = new CreateDatabaseCommand(params);

try {
    const data = await writeClient.send(command);
    console.log(`Database ${data.Database.DatabaseName} created successfully`);
} catch (error) {
    if (error.code === 'ConflictException') {
        console.log(`Database ${params.DatabaseName} already exists. Skipping
 creation.`);
    } else {
        console.log("Error creating database", error);
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function createDatabase() {
    console.log("Creating Database");
    const params = {
        DatabaseName: constants.DATABASE_NAME
```

```
    };

    const promise = writeClient.createDatabase(params).promise();

    await promise.then(
        (data) => {
            console.log(`Database ${data.Database.DatabaseName} created
 successfully`);
        },
        (err) => {
            if (err.code === 'ConflictException') {
                console.log(`Database ${params.DatabaseName} already exists.
 Skipping creation.`);
            } else {
                console.log("Error creating database", err);
            }
        }
    );
}
```

.NET

```
        public async Task CreateDatabase()
        {
            Console.WriteLine("Creating Database");

            try
            {
                var createDatabaseRequest = new CreateDatabaseRequest
                {
                    DatabaseName = Constants.DATABASE_NAME
                };
                CreateDatabaseResponse response = await
 writeClient.CreateDatabaseAsync(createDatabaseRequest);
                Console.WriteLine($"Database {Constants.DATABASE_NAME} created");
            }
            catch (ConflictException)
            {
                Console.WriteLine("Database already exists.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Create database failed:" + e.ToString());
```

```
            }

        }
```

Describe database

You can use the following code snippets to get information about the attributes of your newly created database.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void describeDatabase() {
        System.out.println("Describing database");
        final DescribeDatabaseRequest describeDatabaseRequest = new
DescribeDatabaseRequest();
        describeDatabaseRequest.setDatabaseName(DATABASE_NAME);
        try {
            DescribeDatabaseResult result =
amazonTimestreamWrite.describeDatabase(describeDatabaseRequest);
            final Database databaseRecord = result.getDatabase();
            final String databaseId = databaseRecord.getArn();
            System.out.println("Database " + DATABASE_NAME + " has id " +
databaseId);
        } catch (final Exception e) {
            System.out.println("Database doesn't exist = " + e);
            throw e;
        }
    }
```

Java v2

```
    public void describeDatabase() {
        System.out.println("Describing database");
        final DescribeDatabaseRequest describeDatabaseRequest =
DescribeDatabaseRequest.builder()
```

```
                .databaseName(DATABASE_NAME).build();
        try {
            DescribeDatabaseResponse response =
 timestreamWriteClient.describeDatabase(describeDatabaseRequest);
            final Database databaseRecord = response.database();
            final String databaseId = databaseRecord.arn();
            System.out.println("Database " + DATABASE_NAME + " has id " +
 databaseId);
        } catch (final Exception e) {
            System.out.println("Database doesn't exist = " + e);
            throw e;
        }
    }
```

Go

```
describeDatabaseOutput, err := writeSvc.DescribeDatabase(describeDatabaseInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("Describe database is successful, below is the output:")
        fmt.Println(describeDatabaseOutput)
    }
```

Python

```
    def describe_database(self):
        print("Describing database")
        try:
            result =
 self.client.describe_database(DatabaseName=Constant.DATABASE_NAME)
            print("Database [%s] has id [%s]" % (Constant.DATABASE_NAME,
 result['Database']['Arn']))
        except self.client.exceptions.ResourceNotFoundException:
            print("Database doesn't exist")
        except Exception as err:
            print("Describe database failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class DescribeDatabaseCommand and DescribeDatabase.

```
import { TimestreamWriteClient, DescribeDatabaseCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });

const params = {
    DatabaseName: "testDbFromNode"
};

const command = new DescribeDatabaseCommand(params);

try {
    const data = await writeClient.send(command);
    console.log(`Database ${data.Database.DatabaseName} has id
 ${data.Database.Arn}`);
} catch (error) {
    if (error.code === 'ResourceNotFoundException') {
        console.log("Database doesn't exist.");
    } else {
        console.log("Describe database failed.", error);
        throw error;
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function describeDatabase () {
    console.log("Describing Database");
    const params = {
        DatabaseName: constants.DATABASE_NAME
    };

    const promise = writeClient.describeDatabase(params).promise();

    await promise.then(
        (data) => {
```

```
            console.log(`Database ${data.Database.DatabaseName} has id
 ${data.Database.Arn}`);
        },
        (err) => {
            if (err.code === 'ResourceNotFoundException') {
                console.log("Database doesn't exist.");
            } else {
                console.log("Describe database failed.", err);
                throw err;
            }
        }
    );
}
```

.NET

```
        public async Task DescribeDatabase()
        {
            Console.WriteLine("Describing Database");

            try
            {
                var describeDatabaseRequest = new DescribeDatabaseRequest
                {
                    DatabaseName = Constants.DATABASE_NAME
                };
                DescribeDatabaseResponse response = await
 writeClient.DescribeDatabaseAsync(describeDatabaseRequest);
                Console.WriteLine($"Database {Constants.DATABASE_NAME} has id:
{response.Database.Arn}");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine("Database does not exist.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Describe database failed:" + e.ToString());
            }

        }
```

Update database

You can use the following code snippets to update your databases.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void updateDatabase(String kmsId) {
        System.out.println("Updating kmsId to " + kmsId);
        UpdateDatabaseRequest request = new UpdateDatabaseRequest();
        request.setDatabaseName(DATABASE_NAME);
        request.setKmsKeyId(kmsId);
        try {
            UpdateDatabaseResult result =
amazonTimestreamWrite.updateDatabase(request);
            System.out.println("Update Database complete");
        } catch (final ValidationException e) {
            System.out.println("Update database failed:");
            e.printStackTrace();
        } catch (final ResourceNotFoundException e) {
            System.out.println("Database " + DATABASE_NAME + " doesn't exist = " +
e);
        } catch (final Exception e) {
            System.out.println("Could not update Database " + DATABASE_NAME + " = "
+ e);
            throw e;
        }
    }
```

Java v2

```
    public void updateDatabase(String kmsKeyId) {

        if (kmsKeyId == null) {
            System.out.println("Skipping UpdateDatabase because KmsKeyId was not
given");
            return;
```

```
        }

        System.out.println("Updating database");

        UpdateDatabaseRequest request = UpdateDatabaseRequest.builder()
                .databaseName(DATABASE_NAME)
                .kmsKeyId(kmsKeyId)
                .build();
        try {
            timestreamWriteClient.updateDatabase(request);
            System.out.println("Database [" + DATABASE_NAME + "] updated
 successfully with kmsKeyId " + kmsKeyId);
        } catch (ResourceNotFoundException e) {
            System.out.println("Database [" + DATABASE_NAME + "] does not exist.
 Skipping UpdateDatabase");
        } catch (Exception e) {
            System.out.println("UpdateDatabase failed: " + e);
        }
    }
```

Go

```
// Update Database.
        updateDatabaseInput := &timestreamwrite.UpdateDatabaseInput {
            DatabaseName: aws.String(*databaseName),
            KmsKeyId: aws.String(*kmsKeyId),
        }

        updateDatabaseOutput, err := writeSvc.UpdateDatabase(updateDatabaseInput)

        if err != nil {
            fmt.Println("Error:")
            fmt.Println(err)
        } else {
            fmt.Println("Update database is successful, below is the output:")
            fmt.Println(updateDatabaseOutput)
        }
```

Python

```
    def update_database(self, kms_id):
        print("Updating database")
        try:
```

```
            result =
 self.client.update_database(DatabaseName=Constant.DATABASE_NAME, KmsKeyId=kms_id)
            print("Database [%s] was updated to use kms [%s] successfully" %
 (Constant.DATABASE_NAME,

 result['Database']['KmsKeyId']))
        except self.client.exceptions.ResourceNotFoundException:
            print("Database doesn't exist")
        except Exception as err:
            print("Update database failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class UpdateDatabaseCommand and UpdateDatabase.

```
import { TimestreamWriteClient, UpdateDatabaseCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });
let updatedKmsKeyId = "<updatedKmsKeyId>";

const params = {
    DatabaseName: "testDbFromNode",
    KmsKeyId: updatedKmsKeyId
};

const command = new UpdateDatabaseCommand(params);

try {
    const data = await writeClient.send(command);
    console.log(`Database ${data.Database.DatabaseName} updated kmsKeyId to
 ${updatedKmsKeyId}`);
} catch (error) {
    if (error.code === 'ResourceNotFoundException') {
        console.log("Database doesn't exist.");
    } else {
        console.log("Update database failed.", error);
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function updateDatabase(updatedKmsKeyId) {

    if (updatedKmsKeyId === undefined) {
        console.log("Skipping UpdateDatabase; KmsKeyId was not given");
        return;
    }
    console.log("Updating Database");
    const params = {
        DatabaseName: constants.DATABASE_NAME,
        KmsKeyId: updatedKmsKeyId
    }

    const promise = writeClient.updateDatabase(params).promise();

    await promise.then(
        (data) => {
            console.log(`Database ${data.Database.DatabaseName} updated kmsKeyId to
 ${updatedKmsKeyId}`);
        },
        (err) => {
            if (err.code === 'ResourceNotFoundException') {
                console.log("Database doesn't exist.");
            } else {
                console.log("Update database failed.", err);
            }
        }
    );
}
```

.NET

```
        public async Task UpdateDatabase(String updatedKmsKeyId)
        {
            Console.WriteLine("Updating Database");

            try
            {
                var updateDatabaseRequest = new UpdateDatabaseRequest
                {
                    DatabaseName = Constants.DATABASE_NAME,
```

```
                    KmsKeyId = updatedKmsKeyId
                };
                UpdateDatabaseResponse response = await
writeClient.UpdateDatabaseAsync(updateDatabaseRequest);
                Console.WriteLine($"Database {Constants.DATABASE_NAME} updated with
KmsKeyId {updatedKmsKeyId}");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine("Database does not exist.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Update database failed: " + e.ToString());
            }

        }

        private void PrintDatabases(List<Database> databases)
        {
            foreach (Database database in databases)
                Console.WriteLine($"Database:{database.DatabaseName}");
        }
```

Delete database

You can use the following code snippet to delete a database.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void deleteDatabase() {
        System.out.println("Deleting database");
        final DeleteDatabaseRequest deleteDatabaseRequest = new
DeleteDatabaseRequest();
        deleteDatabaseRequest.setDatabaseName(DATABASE_NAME);
```

```
        try {
            DeleteDatabaseResult result =
                    amazonTimestreamWrite.deleteDatabase(deleteDatabaseRequest);
            System.out.println("Delete database status: " +
 result.getSdkHttpMetadata().getHttpStatusCode());
        } catch (final ResourceNotFoundException e) {
            System.out.println("Database " + DATABASE_NAME + " doesn't exist = " +
 e);
            throw e;
        } catch (final Exception e) {
            System.out.println("Could not delete Database " + DATABASE_NAME + " = "
 + e);
            throw e;
        }
    }
```

Java v2

```
    public void deleteDatabase() {
        System.out.println("Deleting database");
        final DeleteDatabaseRequest deleteDatabaseRequest = new
 DeleteDatabaseRequest();
        deleteDatabaseRequest.setDatabaseName(DATABASE_NAME);
        try {
            DeleteDatabaseResult result =
                    amazonTimestreamWrite.deleteDatabase(deleteDatabaseRequest);
            System.out.println("Delete database status: " +
 result.getSdkHttpMetadata().getHttpStatusCode());
        } catch (final ResourceNotFoundException e) {
            System.out.println("Database " + DATABASE_NAME + " doesn't exist = " +
 e);
            throw e;
        } catch (final Exception e) {
            System.out.println("Could not delete Database " + DATABASE_NAME + " = "
 + e);
            throw e;
        }
    }
```

Go

```
deleteDatabaseInput := &timestreamwrite.DeleteDatabaseInput{
        DatabaseName:   aws.String(*databaseName),
```

```
    }

    _, err = writeSvc.DeleteDatabase(deleteDatabaseInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("Database deleted:", *databaseName)
    }
```

Python

```
    def delete_database(self):
        print("Deleting Database")
        try:
            result =
 self.client.delete_database(DatabaseName=Constant.DATABASE_NAME)
            print("Delete database status [%s]" % result['ResponseMetadata']
['HTTPStatusCode'])
        except self.client.exceptions.ResourceNotFoundException:
            print("database [%s] doesn't exist" % Constant.DATABASE_NAME)
        except Exception as err:
            print("Delete database failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class DeleteDatabaseCommand and DeleteDatabase.

```
import { TimestreamWriteClient, DeleteDatabaseCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });

const params = {
    DatabaseName: "testDbFromNode"
};

const command = new DeleteDatabaseCommand(params);

try {
```

```
    const data = await writeClient.send(command);
    console.log("Deleted database");
} catch (error) {
    if (error.code === 'ResourceNotFoundException') {
        console.log(`Database ${params.DatabaseName} doesn't exists.`);
    } else {
        console.log("Delete database failed.", error);
        throw error;
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function deleteDatabase() {
    console.log("Deleting Database");
    const params = {
        DatabaseName: constants.DATABASE_NAME
    };

    const promise = writeClient.deleteDatabase(params).promise();

    await promise.then(
        function (data) {
            console.log("Deleted database");
         },
        function(err) {
            if (err.code === 'ResourceNotFoundException') {
                console.log(`Database ${params.DatabaseName} doesn't exists.`);
            } else {
                console.log("Delete database failed.", err);
                throw err;
            }
        }
    );
}
```

.NET

```
        public async Task DeleteDatabase()
        {
            Console.WriteLine("Deleting database");
            try
```

```
            {
                var deleteDatabaseRequest = new DeleteDatabaseRequest
                {
                    DatabaseName = Constants.DATABASE_NAME
                };
                DeleteDatabaseResponse response = await
writeClient.DeleteDatabaseAsync(deleteDatabaseRequest);
                Console.WriteLine($"Database {Constants.DATABASE_NAME} delete
request status:{response.HttpStatusCode}");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine($"Database {Constants.DATABASE_NAME} does not
exists");
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception while deleting database:" +
e.ToString());
            }
        }
```

List databases

You can use the following code snippets to list your databases.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void listDatabases() {
        System.out.println("Listing databases");
        ListDatabasesRequest request = new ListDatabasesRequest();
        ListDatabasesResult result = amazonTimestreamWrite.listDatabases(request);
        final List<Database> databases = result.getDatabases();
        printDatabases(databases);
```

```
        String nextToken = result.getNextToken();
        while (nextToken != null && !nextToken.isEmpty()) {
            request.setNextToken(nextToken);
            ListDatabasesResult nextResult =
 amazonTimestreamWrite.listDatabases(request);
            final List<Database> nextDatabases = nextResult.getDatabases();
            printDatabases(nextDatabases);
            nextToken = nextResult.getNextToken();
        }
    }

    private void printDatabases(List<Database> databases) {
        for (Database db : databases) {
            System.out.println(db.getDatabaseName());
        }
    }
```

Java v2

```
    public void listDatabases() {
        System.out.println("Listing databases");
        ListDatabasesRequest request =
 ListDatabasesRequest.builder().maxResults(2).build();
        ListDatabasesIterable listDatabasesIterable =
 timestreamWriteClient.listDatabasesPaginator(request);
        for(ListDatabasesResponse listDatabasesResponse : listDatabasesIterable) {
            final List<Database> databases = listDatabasesResponse.databases();
            databases.forEach(database ->
 System.out.println(database.databaseName()));
        }
    }
```

Go

```
// List databases.
    listDatabasesMaxResult := int64(15)

    listDatabasesInput := &timestreamwrite.ListDatabasesInput{
        MaxResults: &listDatabasesMaxResult,
    }

    listDatabasesOutput, err := writeSvc.ListDatabases(listDatabasesInput)
```

```
    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("List databases is successful, below is the output:")
        fmt.Println(listDatabasesOutput)
    }
```

Python

```
    def list_databases(self):
        print("Listing databases")
        try:
            result = self.client.list_databases(MaxResults=5)
            self._print_databases(result['Databases'])
            next_token = result.get('NextToken', None)
            while next_token:
                result = self.client.list_databases(NextToken=next_token,
 MaxResults=5)
                self._print_databases(result['Databases'])
                next_token = result.get('NextToken', None)
        except Exception as err:
            print("List databases failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class ListDatabasesCommand and ListDatabases.

```
import { TimestreamWriteClient, ListDatabasesCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });

const params = {
    MaxResults: 15
};

const command = new ListDatabasesCommand(params);

getDatabasesList(null);
```

```
async function getDatabasesList(nextToken) {
    if (nextToken) {
        params.NextToken = nextToken;
    }

    try {
        const data = await writeClient.send(command);

        data.Databases.forEach(function (database) {
            console.log(database.DatabaseName);
        });

        if (data.NextToken) {
            return getDatabasesList(data.NextToken);
        }
    } catch (error) {
        console.log("Error while listing databases", error);
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function listDatabases() {
    console.log("Listing databases:");
    const databases = await getDatabasesList(null);
    databases.forEach(function(database){
        console.log(database.DatabaseName);
    });
}

function getDatabasesList(nextToken, databases = []) {
    var params = {
        MaxResults: 15
    };

    if(nextToken) {
        params.NextToken = nextToken;
    }

    return writeClient.listDatabases(params).promise()
        .then(
            (data) => {
```

```
                databases.push.apply(databases, data.Databases);
                if (data.NextToken) {
                    return getDatabasesList(data.NextToken, databases);
                } else {
                    return databases;
                }
            },
            (err) => {
                console.log("Error while listing databases", err);
            });
}
```

.NET

```
        public async Task ListDatabases()
        {
            Console.WriteLine("Listing Databases");

            try
            {
                var listDatabasesRequest = new ListDatabasesRequest
                {
                    MaxResults = 5
                };
                ListDatabasesResponse response = await
 writeClient.ListDatabasesAsync(listDatabasesRequest);
                PrintDatabases(response.Databases);
                var nextToken = response.NextToken;
                while (nextToken != null)
                {
                    listDatabasesRequest.NextToken = nextToken;
                    response = await
 writeClient.ListDatabasesAsync(listDatabasesRequest);
                    PrintDatabases(response.Databases);
                    nextToken = response.NextToken;
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("List database failed:" + e.ToString());
            }

        }
```

Create table

Topics

- Memory store writes
- Magnetic store writes

Memory store writes

You can use the following code snippet to create a table that has magnetic store writes disabled, as a result you can only write data into your memory store retention window.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void createTable() {
        System.out.println("Creating table");
        CreateTableRequest createTableRequest = new CreateTableRequest();
        createTableRequest.setDatabaseName(DATABASE_NAME);
        createTableRequest.setTableName(TABLE_NAME);
        final RetentionProperties retentionProperties = new RetentionProperties()
                .withMemoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                .withMagneticStoreRetentionPeriodInDays(CT_TTL_DAYS);
        createTableRequest.setRetentionProperties(retentionProperties);

        try {
            amazonTimestreamWrite.createTable(createTableRequest);
            System.out.println("Table [" + TABLE_NAME + "] successfully created.");
        } catch (ConflictException e) {
            System.out.println("Table [" + TABLE_NAME + "] exists on database [" +
DATABASE_NAME + "] . Skipping database creation");
        }
    }
```

Java v2

```
    public void createTable() {
        System.out.println("Creating table");

        final RetentionProperties retentionProperties =
 RetentionProperties.builder()
                .memoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                .magneticStoreRetentionPeriodInDays(CT_TTL_DAYS).build();
        final CreateTableRequest createTableRequest = CreateTableRequest.builder()

 .databaseName(DATABASE_NAME).tableName(TABLE_NAME).retentionProperties(retentionProperties)

        try {
            timestreamWriteClient.createTable(createTableRequest);
            System.out.println("Table [" + TABLE_NAME + "] successfully created.");
        } catch (ConflictException e) {
            System.out.println("Table [" + TABLE_NAME + "] exists on database [" +
 DATABASE_NAME + "] . Skipping database creation");
        }
    }
```

Go

```
// Create table.
    createTableInput := &timestreamwrite.CreateTableInput{
        DatabaseName: aws.String(*databaseName),
        TableName:    aws.String(*tableName),
    }
    _, err = writeSvc.CreateTable(createTableInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("Create table is successful")
    }
```

Python

```
    def create_table(self):
        print("Creating table")
        retention_properties = {
```

```
            'MemoryStoreRetentionPeriodInHours': Constant.HT_TTL_HOURS,
            'MagneticStoreRetentionPeriodInDays': Constant.CT_TTL_DAYS
        }
        try:
            self.client.create_table(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                                     RetentionProperties=retention_properties)
            print("Table [%s] successfully created." % Constant.TABLE_NAME)
        except self.client.exceptions.ConflictException:
            print("Table [%s] exists on database [%s]. Skipping table creation" % (
                Constant.TABLE_NAME, Constant.DATABASE_NAME))
        except Exception as err:
            print("Create table failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class CreateTableCommand and CreateTable.

```
import { TimestreamWriteClient, CreateTableCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });

const params = {
    DatabaseName: "testDbFromNode",
    TableName: "testTableFromNode",
    RetentionProperties: {
        MemoryStoreRetentionPeriodInHours: 24,
        MagneticStoreRetentionPeriodInDays: 365
    }
};

const command = new CreateTableCommand(params);

try {
    const data = await writeClient.send(command);
    console.log(`Table ${data.Table.TableName} created successfully`);
} catch (error) {
    if (error.code === 'ConflictException') {
        console.log(`Table ${params.TableName} already exists on db
 ${params.DatabaseName}. Skipping creation.`);
```

```
    } else {
        console.log("Error creating table. ", error);
        throw error;
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function createTable() {
    console.log("Creating Table");
    const params = {
        DatabaseName: constants.DATABASE_NAME,
        TableName: constants.TABLE_NAME,
        RetentionProperties: {
            MemoryStoreRetentionPeriodInHours: constants.HT_TTL_HOURS,
            MagneticStoreRetentionPeriodInDays: constants.CT_TTL_DAYS
        }
    };

    const promise = writeClient.createTable(params).promise();

    await promise.then(
        (data) => {
            console.log(`Table ${data.Table.TableName} created successfully`);
        },
        (err) => {
            if (err.code === 'ConflictException') {
                console.log(`Table ${params.TableName} already exists on db
 ${params.DatabaseName}. Skipping creation.`);
            } else {
                console.log("Error creating table. ", err);
                throw err;
            }
        }
    );
}
```

.NET

```
        public async Task CreateTable()
        {
            Console.WriteLine("Creating Table");
```

```
            try
            {
                var createTableRequest = new CreateTableRequest
                {
                    DatabaseName = Constants.DATABASE_NAME,
                    TableName = Constants.TABLE_NAME,
                    RetentionProperties = new RetentionProperties
                    {
                        MagneticStoreRetentionPeriodInDays = Constants.CT_TTL_DAYS,
                        MemoryStoreRetentionPeriodInHours = Constants.HT_TTL_HOURS
                    }
                };
                CreateTableResponse response = await
writeClient.CreateTableAsync(createTableRequest);
                Console.WriteLine($"Table {Constants.TABLE_NAME} created");
            }
            catch (ConflictException)
            {
                Console.WriteLine("Table already exists.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Create table failed:" + e.ToString());
            }

        }
```

Magnetic store writes

You can use the following code snippet to create a table with magnetic store writes enabled. With magnetic store writes you can write data into both your memory store retention window and magnetic store retention window.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void createTable(String databaseName, String tableName) {
        System.out.println("Creating table");
        CreateTableRequest createTableRequest = new CreateTableRequest();
        createTableRequest.setDatabaseName(databaseName);
        createTableRequest.setTableName(tableName);
        final RetentionProperties retentionProperties = new RetentionProperties()
                .withMemoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                .withMagneticStoreRetentionPeriodInDays(CT_TTL_DAYS);
        createTableRequest.setRetentionProperties(retentionProperties);
        // Enable MagneticStoreWrite
        final MagneticStoreWriteProperties magneticStoreWriteProperties = new
MagneticStoreWriteProperties()
                .withEnableMagneticStoreWrites(true);

createTableRequest.setMagneticStoreWriteProperties(magneticStoreWriteProperties);
        try {
            amazonTimestreamWrite.createTable(createTableRequest);
            System.out.println("Table [" + tableName + "] successfully created.");
        } catch (ConflictException e) {
            System.out.println("Table [" + tableName + "] exists on database [" +
databaseName + "] . Skipping table creation");
            //We do not throw exception here, we use the existing table instead
        }
    }
```

Java v2

```
    public void createTable(String databaseName, String tableName) {
        System.out.println("Creating table");

        // Enable MagneticStoreWrite
        final MagneticStoreWriteProperties magneticStoreWriteProperties =
                MagneticStoreWriteProperties.builder()
                        .enableMagneticStoreWrites(true)
                        .build();

        CreateTableRequest createTableRequest =
                CreateTableRequest.builder()
                        .databaseName(databaseName)
                        .tableName(tableName)
                        .retentionProperties(RetentionProperties.builder()
```

```
                                .memoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                                .magneticStoreRetentionPeriodInDays(CT_TTL_DAYS)
                                .build())
                        .magneticStoreWriteProperties(magneticStoreWriteProperties)
                        .build();
        try {
            timestreamWriteClient.createTable(createTableRequest);
            System.out.println("Table [" + tableName + "] successfully created.");
        } catch (ConflictException e) {
            System.out.println("Table [" + tableName + "] exists in database [" +
 databaseName + "] . Skipping table creation");
        }
    }
```

Go

```
// Create table.
    createTableInput := &timestreamwrite.CreateTableInput{
        DatabaseName: aws.String(*databaseName),
        TableName:    aws.String(*tableName),
    // Enable MagneticStoreWrite
        MagneticStoreWriteProperties: &timestreamwrite.MagneticStoreWriteProperties{
            EnableMagneticStoreWrites: aws.Bool(true),
             },
      }
    _, err = writeSvc.CreateTable(createTableInput)
```

Python

```
    def create_table(self):
        print("Creating table")
        retention_properties = {
            'MemoryStoreRetentionPeriodInHours': Constant.HT_TTL_HOURS,
            'MagneticStoreRetentionPeriodInDays': Constant.CT_TTL_DAYS
        }
        magnetic_store_write_properties = {
            'EnableMagneticStoreWrites': True
        }
        try:
            self.client.create_table(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                                     RetentionProperties=retention_properties,
```

```
 MagneticStoreWriteProperties=magnetic_store_write_properties)
            print("Table [%s] successfully created." % Constant.TABLE_NAME)
        except self.client.exceptions.ConflictException:
            print("Table [%s] exists on database [%s]. Skipping table creation" % (
                Constant.TABLE_NAME, Constant.DATABASE_NAME))
        except Exception as err:
            print("Create table failed:", err)
```

Node.js

```
async function createTable() {
    console.log("Creating Table");

    const params = {
        DatabaseName: constants.DATABASE_NAME,
        TableName: constants.TABLE_NAME,
        RetentionProperties: {
            MemoryStoreRetentionPeriodInHours: constants.HT_TTL_HOURS,
            MagneticStoreRetentionPeriodInDays: constants.CT_TTL_DAYS
        },
        MagneticStoreWriteProperties: {
            EnableMagneticStoreWrites: true
        }
    };

    const promise = writeClient.createTable(params).promise();

    await promise.then(
        (data) => {
            console.log(`Table ${data.Table.TableName} created successfully`);
        },
        (err) => {
            if (err.code === 'ConflictException') {
                console.log(`Table ${params.TableName} already exists on db
 ${params.DatabaseName}. Skipping creation.`);
            } else {
                console.log("Error creating table. ", err);
                throw err;
            }
        }
    );
}
```

.NET

```
        public async Task CreateTable()
        {
            Console.WriteLine("Creating Table");

            try
            {
                var createTableRequest = new CreateTableRequest
                {
                    DatabaseName = Constants.DATABASE_NAME,
                    TableName = Constants.TABLE_NAME,
                    RetentionProperties = new RetentionProperties
                    {
                        MagneticStoreRetentionPeriodInDays = Constants.CT_TTL_DAYS,
                        MemoryStoreRetentionPeriodInHours = Constants.HT_TTL_HOURS
                    },
                    // Enable MagneticStoreWrite
                    MagneticStoreWriteProperties = new MagneticStoreWriteProperties
                    {
                        EnableMagneticStoreWrites = true,
                    }
                };
                CreateTableResponse response = await
writeClient.CreateTableAsync(createTableRequest);
                Console.WriteLine($"Table {Constants.TABLE_NAME} created");
            }
            catch (ConflictException)
            {
                Console.WriteLine("Table already exists.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Create table failed:" + e.ToString());
            }

        }
```

Describe table

You can use the following code snippets to get information about the attributes of your table.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
   public void describeTable() {
       System.out.println("Describing table");
       final DescribeTableRequest describeTableRequest = new
DescribeTableRequest();
       describeTableRequest.setDatabaseName(DATABASE_NAME);
       describeTableRequest.setTableName(TABLE_NAME);
       try {
           DescribeTableResult result =
amazonTimestreamWrite.describeTable(describeTableRequest);
           String tableId = result.getTable().getArn();
           System.out.println("Table " + TABLE_NAME + " has id " + tableId);
       } catch (final Exception e) {
           System.out.println("Table " + TABLE_NAME + " doesn't exist = " + e);
           throw e;
       }
   }
```

Java v2

```
   public void describeTable() {
       System.out.println("Describing table");
       final DescribeTableRequest describeTableRequest =
DescribeTableRequest.builder()
               .databaseName(DATABASE_NAME).tableName(TABLE_NAME).build();
       try {
           DescribeTableResponse response =
timestreamWriteClient.describeTable(describeTableRequest);
           String tableId = response.table().arn();
           System.out.println("Table " + TABLE_NAME + " has id " + tableId);
       } catch (final Exception e) {
           System.out.println("Table " + TABLE_NAME + " doesn't exist = " + e);
           throw e;
       }
   }
```

Go

```
// Describe table.
    describeTableInput := &timestreamwrite.DescribeTableInput{
        DatabaseName: aws.String(*databaseName),
        TableName:    aws.String(*tableName),
    }
    describeTableOutput, err := writeSvc.DescribeTable(describeTableInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("Describe table is successful, below is the output:")
        fmt.Println(describeTableOutput)
    }
```

Python

```
    def describe_table(self):
        print("Describing table")
        try:
            result = self.client.describe_table(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME)
            print("Table [%s] has id [%s]" % (Constant.TABLE_NAME, result['Table']
['Arn']))
        except self.client.exceptions.ResourceNotFoundException:
            print("Table doesn't exist")
        except Exception as err:
            print("Describe table failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class DescribeTableCommand and DescribeTable.

```
import { TimestreamWriteClient, DescribeTableCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });
```

```
const params = {
    DatabaseName: "testDbFromNode",
    TableName: "testTableFromNode"
};

const command = new DescribeTableCommand(params);

try {
    const data = await writeClient.send(command);
    console.log(`Table ${data.Table.TableName} has id ${data.Table.Arn}`);
} catch (error) {
    if (error.code === 'ResourceNotFoundException') {
        console.log("Table or Database doesn't exist.");
    } else {
        console.log("Describe table failed.", error);
        throw error;
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function describeTable() {
    console.log("Describing Table");
    const params = {
        DatabaseName: constants.DATABASE_NAME,
        TableName: constants.TABLE_NAME
    };

    const promise = writeClient.describeTable(params).promise();

    await promise.then(
        (data) => {
            console.log(`Table ${data.Table.TableName} has id ${data.Table.Arn}`);
        },
        (err) => {
            if (err.code === 'ResourceNotFoundException') {
                console.log("Table or Database doesn't exists.");
            } else {
                console.log("Describe table failed.", err);
                throw err;
            }
        }
```

```
    );
}
```

.NET

```
        public async Task DescribeTable()
        {
            Console.WriteLine("Describing Table");

            try
            {
                var describeTableRequest = new DescribeTableRequest
                {
                    DatabaseName = Constants.DATABASE_NAME,
                    TableName = Constants.TABLE_NAME
                };
                DescribeTableResponse response = await
 writeClient.DescribeTableAsync(describeTableRequest);
                Console.WriteLine($"Table {Constants.TABLE_NAME} has id:
{response.Table.Arn}");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine("Table does not exist.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Describe table failed:" + e.ToString());
            }

        }
```

Update table

You can use the following code snippets to update a table.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void updateTable() {
        System.out.println("Updating table");
        UpdateTableRequest updateTableRequest = new UpdateTableRequest();
        updateTableRequest.setDatabaseName(DATABASE_NAME);
        updateTableRequest.setTableName(TABLE_NAME);

        final RetentionProperties retentionProperties = new RetentionProperties()
                .withMemoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                .withMagneticStoreRetentionPeriodInDays(CT_TTL_DAYS);

        updateTableRequest.setRetentionProperties(retentionProperties);

        amazonTimestreamWrite.updateTable(updateTableRequest);
        System.out.println("Table updated");
    }
```

Java v2

```
    public void updateTable() {
        System.out.println("Updating table");

        final RetentionProperties retentionProperties =
 RetentionProperties.builder()
                .memoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                .magneticStoreRetentionPeriodInDays(CT_TTL_DAYS).build();
        final UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()

 .databaseName(DATABASE_NAME).tableName(TABLE_NAME).retentionProperties(retentionProperties)

        timestreamWriteClient.updateTable(updateTableRequest);
        System.out.println("Table updated");
    }
```

Go

```
// Update table.
    magneticStoreRetentionPeriodInDays := int64(7 * 365)
    memoryStoreRetentionPeriodInHours := int64(24)

    updateTableInput := &timestreamwrite.UpdateTableInput{
        DatabaseName: aws.String(*databaseName),
```

```
        TableName:    aws.String(*tableName),
        RetentionProperties: &timestreamwrite.RetentionProperties{
            MagneticStoreRetentionPeriodInDays: &magneticStoreRetentionPeriodInDays,
            MemoryStoreRetentionPeriodInHours:  &memoryStoreRetentionPeriodInHours,
        },
    }
    updateTableOutput, err := writeSvc.UpdateTable(updateTableInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("Update table is successful, below is the output:")
        fmt.Println(updateTableOutput)
    }
```

Python

```
    def update_table(self):
        print("Updating table")
        retention_properties = {
            'MemoryStoreRetentionPeriodInHours': Constant.HT_TTL_HOURS,
            'MagneticStoreRetentionPeriodInDays': Constant.CT_TTL_DAYS
        }
        try:
            self.client.update_table(DatabaseName=Constant.DATABASE_NAME,
TableName=Constant.TABLE_NAME,
                                     RetentionProperties=retention_properties)
            print("Table updated.")
        except Exception as err:
            print("Update table failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class UpdateTableCommand and UpdateTable.

```
import { TimestreamWriteClient, UpdateTableCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });
```

```
const params = {
    DatabaseName: "testDbFromNode",
    TableName: "testTableFromNode",
    RetentionProperties: {
        MemoryStoreRetentionPeriodInHours: 24,
        MagneticStoreRetentionPeriodInDays: 180
    }
};

const command = new UpdateTableCommand(params);

try {
    const data = await writeClient.send(command);
    console.log("Table updated")
} catch (error) {
    console.log("Error updating table. ", error);
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function updateTable() {
    console.log("Updating Table");
    const params = {
        DatabaseName: constants.DATABASE_NAME,
        TableName: constants.TABLE_NAME,
        RetentionProperties: {
            MemoryStoreRetentionPeriodInHours: constants.HT_TTL_HOURS,
            MagneticStoreRetentionPeriodInDays: constants.CT_TTL_DAYS
        }
    };

    const promise = writeClient.updateTable(params).promise();

    await promise.then(
        (data) => {
            console.log("Table updated")
        },
        (err) => {
            console.log("Error updating table. ", err);
            throw err;
        }
    );
```

```
}
```

.NET

```
        public async Task UpdateTable()
        {
            Console.WriteLine("Updating Table");

            try
            {
                var updateTableRequest = new UpdateTableRequest
                {
                    DatabaseName = Constants.DATABASE_NAME,
                    TableName = Constants.TABLE_NAME,
                    RetentionProperties = new RetentionProperties
                    {
                        MagneticStoreRetentionPeriodInDays = Constants.CT_TTL_DAYS,
                        MemoryStoreRetentionPeriodInHours = Constants.HT_TTL_HOURS
                    }
                };
                UpdateTableResponse response = await
writeClient.UpdateTableAsync(updateTableRequest);
                Console.WriteLine($"Table {Constants.TABLE_NAME} updated");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine("Table does not exist.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Update table failed:" + e.ToString());
            }

        }
```

Delete table

You can use the following code snippets to delete a table.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void deleteTable() {
        System.out.println("Deleting table");
        final DeleteTableRequest deleteTableRequest = new DeleteTableRequest();
        deleteTableRequest.setDatabaseName(DATABASE_NAME);
        deleteTableRequest.setTableName(TABLE_NAME);
        try {
            DeleteTableResult result =
                    amazonTimestreamWrite.deleteTable(deleteTableRequest);
            System.out.println("Delete table status: " +
result.getSdkHttpMetadata().getHttpStatusCode());
        } catch (final ResourceNotFoundException e) {
            System.out.println("Table " + TABLE_NAME + " doesn't exist = " + e);
            throw e;
        } catch (final Exception e) {
            System.out.println("Could not delete table " + TABLE_NAME + " = " + e);
            throw e;
        }
    }
```

Java v2

```
    public void deleteTable() {
        System.out.println("Deleting table");
        final DeleteTableRequest deleteTableRequest = DeleteTableRequest.builder()
                .databaseName(DATABASE_NAME).tableName(TABLE_NAME).build();
        try {
            DeleteTableResponse response =
                    timestreamWriteClient.deleteTable(deleteTableRequest);
            System.out.println("Delete table status: " +
response.sdkHttpResponse().statusCode());
        } catch (final ResourceNotFoundException e) {
            System.out.println("Table " + TABLE_NAME + " doesn't exist = " + e);
            throw e;
        } catch (final Exception e) {
```

```
            System.out.println("Could not delete table " + TABLE_NAME + " = " + e);
            throw e;
        }
    }
```

Go

```
deleteTableInput := &timestreamwrite.DeleteTableInput{
        DatabaseName:   aws.String(*databaseName),
        TableName:    aws.String(*tableName),
    }
    _, err = writeSvc.DeleteTable(deleteTableInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("Table deleted", *tableName)
    }
```

Python

```
    def delete_table(self):
        print("Deleting Table")
        try:
            result = self.client.delete_table(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME)
            print("Delete table status [%s]" % result['ResponseMetadata']
['HTTPStatusCode'])
        except self.client.exceptions.ResourceNotFoundException:
            print("Table [%s] doesn't exist" % Constant.TABLE_NAME)
        except Exception as err:
            print("Delete table failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class DeleteTableCommand and DeleteTable.

```
import { TimestreamWriteClient, DeleteTableCommand } from "@aws-sdk/client-
timestream-write";
```

```
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });

const params = {
    DatabaseName: "testDbFromNode",
    TableName: "testTableFromNode"
};

const command = new DeleteTableCommand(params);

try {
    const data = await writeClient.send(command);
    console.log("Deleted table");
} catch (error) {
    if (error.code === 'ResourceNotFoundException') {
        console.log(`Table ${params.TableName} or Database ${params.DatabaseName}
 doesn't exist.`);
    } else {
        console.log("Delete table failed.", error);
        throw error;
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function deleteTable() {
    console.log("Deleting Table");
    const params = {
        DatabaseName: constants.DATABASE_NAME,
        TableName: constants.TABLE_NAME
    };

    const promise = writeClient.deleteTable(params).promise();

    await promise.then(
        function (data) {
            console.log("Deleted table");
        },
        function(err) {
            if (err.code === 'ResourceNotFoundException') {
                console.log(`Table ${params.TableName} or Database
 ${params.DatabaseName} doesn't exists.`);
            } else {
```

```
                console.log("Delete table failed.", err);
                throw err;
            }
        }
    );
}
```

.NET

```
        public async Task DeleteTable()
        {
            Console.WriteLine("Deleting table");
            try
            {
                var deleteTableRequest = new DeleteTableRequest
                {
                    DatabaseName = Constants.DATABASE_NAME,
                    TableName = Constants.TABLE_NAME
                };
                DeleteTableResponse response = await
 writeClient.DeleteTableAsync(deleteTableRequest);
                Console.WriteLine($"Table {Constants.TABLE_NAME} delete request
 status: {response.HttpStatusCode}");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine($"Table {Constants.TABLE_NAME} does not exists");
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception while deleting table:" + e.ToString());
            }
        }
```

List tables

You can use the following code snippets to list tables.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void listTables() {
        System.out.println("Listing tables");
        ListTablesRequest request = new ListTablesRequest();
        request.setDatabaseName(DATABASE_NAME);
        ListTablesResult result = amazonTimestreamWrite.listTables(request);
        printTables(result.getTables());

        String nextToken = result.getNextToken();
        while (nextToken != null && !nextToken.isEmpty()) {
            request.setNextToken(nextToken);
            ListTablesResult nextResult = amazonTimestreamWrite.listTables(request);

            printTables(nextResult.getTables());
            nextToken = nextResult.getNextToken();
        }
    }

     private void printTables(List<Table> tables) {
        for (Table table : tables) {
            System.out.println(table.getTableName());
        }
    }
```

Java v2

```
    public void listTables() {
        System.out.println("Listing tables");
        ListTablesRequest request =
ListTablesRequest.builder().databaseName(DATABASE_NAME).maxResults(2).build();
        ListTablesIterable listTablesIterable =
timestreamWriteClient.listTablesPaginator(request);
        for(ListTablesResponse listTablesResponse : listTablesIterable) {
            final List<Table> tables = listTablesResponse.tables();
            tables.forEach(table -> System.out.println(table.tableName()));
```

```
        }
    }
```

Go

```
listTablesMaxResult := int64(15)

    listTablesInput := &timestreamwrite.ListTablesInput{
        DatabaseName: aws.String(*databaseName),
        MaxResults:   &listTablesMaxResult,
    }
    listTablesOutput, err := writeSvc.ListTables(listTablesInput)

    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    } else {
        fmt.Println("List tables is successful, below is the output:")
        fmt.Println(listTablesOutput)
    }
```

Python

```
    def list_tables(self):
        print("Listing tables")
        try:
            result = self.client.list_tables(DatabaseName=Constant.DATABASE_NAME,
 MaxResults=5)
            self.__print_tables(result['Tables'])
            next_token = result.get('NextToken', None)
            while next_token:
                result =
 self.client.list_tables(DatabaseName=Constant.DATABASE_NAME,
                                                    NextToken=next_token, MaxResults=5)
                self.__print_tables(result['Tables'])
                next_token = result.get('NextToken', None)
        except Exception as err:
            print("List tables failed:", err)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

Also see Class ListTablesCommand and ListTables.

```
import { TimestreamWriteClient, ListTablesCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-east-1" });

const params = {
    DatabaseName: "testDbFromNode",
    MaxResults: 15
};

const command = new ListTablesCommand(params);

getTablesList(null);

async function getTablesList(nextToken) {
    if (nextToken) {
        params.NextToken = nextToken;
    }

    try {
        const data = await writeClient.send(command);

        data.Tables.forEach(function (table) {
            console.log(table.TableName);
        });

        if (data.NextToken) {
            return getTablesList(data.NextToken);
        }
    } catch (error) {
        console.log("Error while listing tables", error);
    }
}
```

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function listTables() {
    console.log("Listing tables:");
    const tables = await getTablesList(null);
    tables.forEach(function(table){
        console.log(table.TableName);
```

```
    });
}

function getTablesList(nextToken, tables = []) {
    var params = {
        DatabaseName: constants.DATABASE_NAME,
        MaxResults: 15
    };

    if(nextToken) {
        params.NextToken = nextToken;
    }

    return writeClient.listTables(params).promise()
        .then(
            (data) => {
                tables.push.apply(tables, data.Tables);
                if (data.NextToken) {
                    return getTablesList(data.NextToken, tables);
                } else {
                    return tables;
                }
            },
            (err) => {
                console.log("Error while listing databases", err);
            });
}
```

.NET

```
        public async Task ListTables()
        {
            Console.WriteLine("Listing Tables");

            try
            {
                var listTablesRequest = new ListTablesRequest
                {
                    MaxResults = 5,
                    DatabaseName = Constants.DATABASE_NAME
                };
                ListTablesResponse response = await
 writeClient.ListTablesAsync(listTablesRequest);
```

```
                PrintTables(response.Tables);
                string nextToken = response.NextToken;
                while (nextToken != null)
                {
                    listTablesRequest.NextToken = nextToken;
                    response = await writeClient.ListTablesAsync(listTablesRequest);
                    PrintTables(response.Tables);
                    nextToken = response.NextToken;
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("List table failed:" + e.ToString());
            }

        }

        private void PrintTables(List<Table> tables)
        {
            foreach (Table table in tables)
                Console.WriteLine($"Table: {table.TableName}");
        }
```

Write data (inserts and upserts)

Topics

- Writing batches of records
- Writing batches of records with common attributes
- Upserting records
- Multi-measure attribute example
- Handling write failures

Writing batches of records

You can use the following code snippets to write data into an Amazon Timestream table. Writing data in batches helps to optimize the cost of writes. See Calculating the number of writes for more information.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
  public void writeRecords() {
    System.out.println("Writing records");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region = new Dimension().withName("region").withValue("us-
east-1");
    final Dimension az = new Dimension().withName("az").withValue("az1");
    final Dimension hostname = new
 Dimension().withName("hostname").withValue("host1");

    dimensions.add(region);
    dimensions.add(az);
    dimensions.add(hostname);

    Record cpuUtilization = new Record()
        .withDimensions(dimensions)
        .withMeasureName("cpu_utilization")
        .withMeasureValue("13.5")
        .withMeasureValueType(MeasureValueType.DOUBLE)
        .withTime(String.valueOf(time));
    Record memoryUtilization = new Record()
        .withDimensions(dimensions)
        .withMeasureName("memory_utilization")
        .withMeasureValue("40")
        .withMeasureValueType(MeasureValueType.DOUBLE)
        .withTime(String.valueOf(time));

    records.add(cpuUtilization);
    records.add(memoryUtilization);

    WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
        .withDatabaseName(DATABASE_NAME)
```

```
        .withTableName(TABLE_NAME)
        .withRecords(records);

    try {
      WriteRecordsResult writeRecordsResult =
 amazonTimestreamWrite.writeRecords(writeRecordsRequest);
      System.out.println("WriteRecords Status: " +
 writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("RejectedRecords: " + e);
      for (RejectedRecord rejectedRecord : e.getRejectedRecords()) {
        System.out.println("Rejected Index " + rejectedRecord.getRecordIndex() + ":
 "
            + rejectedRecord.getReason());
      }
      System.out.println("Other records were written successfully. ");
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

Java v2

```
  public void writeRecords() {
    System.out.println("Writing records");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region = Dimension.builder().name("region").value("us-
east-1").build();
    final Dimension az = Dimension.builder().name("az").value("az1").build();
    final Dimension hostname =
 Dimension.builder().name("hostname").value("host1").build();

    dimensions.add(region);
    dimensions.add(az);
    dimensions.add(hostname);

    Record cpuUtilization = Record.builder()
        .dimensions(dimensions)
        .measureValueType(MeasureValueType.DOUBLE)
```

```
        .measureName("cpu_utilization")
        .measureValue("13.5")
        .time(String.valueOf(time)).build();

    Record memoryUtilization = Record.builder()
        .dimensions(dimensions)
        .measureValueType(MeasureValueType.DOUBLE)
        .measureName("memory_utilization")
        .measureValue("40")
        .time(String.valueOf(time)).build();

    records.add(cpuUtilization);
    records.add(memoryUtilization);

    WriteRecordsRequest writeRecordsRequest = WriteRecordsRequest.builder()
        .databaseName(DATABASE_NAME).tableName(TABLE_NAME).records(records).build();

    try {
      WriteRecordsResponse writeRecordsResponse =
 timestreamWriteClient.writeRecords(writeRecordsRequest);
      System.out.println("WriteRecords Status: " +
 writeRecordsResponse.sdkHttpResponse().statusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("RejectedRecords: " + e);
      for (RejectedRecord rejectedRecord : e.rejectedRecords()) {
        System.out.println("Rejected Index " + rejectedRecord.recordIndex() + ": "
            + rejectedRecord.reason());
      }
      System.out.println("Other records were written successfully. ");
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

Go

```
now := time.Now()
currentTimeInSeconds := now.Unix()
writeRecordsInput := &timestreamwrite.WriteRecordsInput{
  DatabaseName: aws.String(*databaseName),
  TableName:  aws.String(*tableName),
  Records: []*timestreamwrite.Record{
    &timestreamwrite.Record{
```

```
      Dimensions: []*timestreamwrite.Dimension{
        &timestreamwrite.Dimension{
          Name:  aws.String("region"),
          Value: aws.String("us-east-1"),
        },
        &timestreamwrite.Dimension{
          Name:  aws.String("az"),
          Value: aws.String("az1"),
        },
        &timestreamwrite.Dimension{
          Name:  aws.String("hostname"),
          Value: aws.String("host1"),
        },
      },
      MeasureName:    aws.String("cpu_utilization"),
      MeasureValue:   aws.String("13.5"),
      MeasureValueType: aws.String("DOUBLE"),
      Time:       aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
      TimeUnit:  aws.String("SECONDS"),
    },
    &timestreamwrite.Record{
      Dimensions: []*timestreamwrite.Dimension{
        &timestreamwrite.Dimension{
          Name:  aws.String("region"),
          Value: aws.String("us-east-1"),
        },
        &timestreamwrite.Dimension{
          Name:  aws.String("az"),
          Value: aws.String("az1"),
        },
        &timestreamwrite.Dimension{
          Name:  aws.String("hostname"),
          Value: aws.String("host1"),
        },
      },
      MeasureName:    aws.String("memory_utilization"),
      MeasureValue:   aws.String("40"),
      MeasureValueType: aws.String("DOUBLE"),
      Time:       aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
      TimeUnit:  aws.String("SECONDS"),
    },
  },
}
```

```
_, err = writeSvc.WriteRecords(writeRecordsInput)

if err != nil {
  fmt.Println("Error:")
  fmt.Println(err)
} else {
  fmt.Println("Write records is successful")
}
```

Python

```
  def write_records(self):
    print("Writing records")
    current_time = self._current_milli_time()

    dimensions = [
      {'Name': 'region', 'Value': 'us-east-1'},
      {'Name': 'az', 'Value': 'az1'},
      {'Name': 'hostname', 'Value': 'host1'}
    ]

    cpu_utilization = {
      'Dimensions': dimensions,
      'MeasureName': 'cpu_utilization',
      'MeasureValue': '13.5',
      'MeasureValueType': 'DOUBLE',
      'Time': current_time
    }

    memory_utilization = {
      'Dimensions': dimensions,
      'MeasureName': 'memory_utilization',
      'MeasureValue': '40',
      'MeasureValueType': 'DOUBLE',
      'Time': current_time
    }

    records = [cpu_utilization, memory_utilization]

    try:
      result = self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                         Records=records, CommonAttributes={})
```

```
      print("WriteRecords Status: [%s]" % result['ResponseMetadata']
['HTTPStatusCode'])
    except self.client.exceptions.RejectedRecordsException as err:
      self._print_rejected_records_exceptions(err)
    except Exception as err:
      print("Error:", err)

  @staticmethod
  def _print_rejected_records_exceptions(err):
    print("RejectedRecords: ", err)
    for rr in err.response["RejectedRecords"]:
      print("Rejected Index " + str(rr["RecordIndex"]) + ": " + rr["Reason"])
      if "ExistingVersion" in rr:
        print("Rejected record existing version: ", rr["ExistingVersion"])

  @staticmethod
  def _current_milli_time():
    return str(int(round(time.time() * 1000)))
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function writeRecords() {
  console.log("Writing records");
  const currentTime = Date.now().toString(); // Unix time in milliseconds

  const dimensions = [
    {'Name': 'region', 'Value': 'us-east-1'},
    {'Name': 'az', 'Value': 'az1'},
    {'Name': 'hostname', 'Value': 'host1'}
  ];

  const cpuUtilization = {
    'Dimensions': dimensions,
    'MeasureName': 'cpu_utilization',
    'MeasureValue': '13.5',
    'MeasureValueType': 'DOUBLE',
    'Time': currentTime.toString()
  };

  const memoryUtilization = {
```

```
    'Dimensions': dimensions,
    'MeasureName': 'memory_utilization',
    'MeasureValue': '40',
    'MeasureValueType': 'DOUBLE',
    'Time': currentTime.toString()
  };

  const records = [cpuUtilization, memoryUtilization];

  const params = {
    DatabaseName: constants.DATABASE_NAME,
    TableName: constants.TABLE_NAME,
    Records: records
  };

  const request = writeClient.writeRecords(params);

  await request.promise().then(
    (data) => {
      console.log("Write records successful");
    },
    (err) => {
      console.log("Error writing records:", err);
      if (err.code === 'RejectedRecordsException') {
        const responsePayload =
 JSON.parse(request.response.httpResponse.body.toString());
        console.log("RejectedRecords: ", responsePayload.RejectedRecords);
        console.log("Other records were written successfully. ");
      }
    }
  );
}
```

.NET

```
   public async Task WriteRecords()
   {
     Console.WriteLine("Writing records");

     DateTimeOffset now = DateTimeOffset.UtcNow;
     string currentTimeString = (now.ToUnixTimeMilliseconds()).ToString();

     List<Dimension> dimensions = new List<Dimension>{
```

```
      new Dimension { Name = "region", Value = "us-east-1" },
      new Dimension { Name = "az", Value = "az1" },
      new Dimension { Name = "hostname", Value = "host1" }
    };

    var cpuUtilization = new Record
    {
      Dimensions = dimensions,
      MeasureName = "cpu_utilization",
      MeasureValue = "13.6",
      MeasureValueType = MeasureValueType.DOUBLE,
      Time = currentTimeString
    };

    var memoryUtilization = new Record
    {
      Dimensions = dimensions,
      MeasureName = "memory_utilization",
      MeasureValue = "40",
      MeasureValueType = MeasureValueType.DOUBLE,
      Time = currentTimeString
    };


    List<Record> records = new List<Record> {
      cpuUtilization,
      memoryUtilization
    };

    try
    {
      var writeRecordsRequest = new WriteRecordsRequest
      {
        DatabaseName = Constants.DATABASE_NAME,
        TableName = Constants.TABLE_NAME,
        Records = records
      };
      WriteRecordsResponse response = await
writeClient.WriteRecordsAsync(writeRecordsRequest);
      Console.WriteLine($"Write records status code:
{response.HttpStatusCode.ToString()}");
    }
    catch (RejectedRecordsException e) {
      Console.WriteLine("RejectedRecordsException:" + e.ToString());
```

```
        foreach (RejectedRecord rr in e.RejectedRecords) {
          Console.WriteLine("RecordIndex " + rr.RecordIndex + " : " + rr.Reason);
        }
        Console.WriteLine("Other records were written successfully. ");
      }
      catch (Exception e)
      {
        Console.WriteLine("Write records failure:" + e.ToString());
      }
    }
```

Writing batches of records with common attributes

If your time series data has measures and/or dimensions that are common across many data points, you can also use the following optimized version of the writeRecords API to insert data into Timestream for LiveAnalytics. Using common attributes with batching can further optimize the cost of writes as described in Calculating the number of writes.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
  public void writeRecordsWithCommonAttributes() {
    System.out.println("Writing records with extracting common attributes");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region = new Dimension().withName("region").withValue("us-
east-1");
    final Dimension az = new Dimension().withName("az").withValue("az1");
    final Dimension hostname = new
 Dimension().withName("hostname").withValue("host1");

    dimensions.add(region);
    dimensions.add(az);
```

```
    dimensions.add(hostname);

    Record commonAttributes = new Record()
        .withDimensions(dimensions)
        .withMeasureValueType(MeasureValueType.DOUBLE)
        .withTime(String.valueOf(time));

    Record cpuUtilization = new Record()
        .withMeasureName("cpu_utilization")
        .withMeasureValue("13.5");
    Record memoryUtilization = new Record()
        .withMeasureName("memory_utilization")
        .withMeasureValue("40");

    records.add(cpuUtilization);
    records.add(memoryUtilization);

    WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
        .withDatabaseName(DATABASE_NAME)
        .withTableName(TABLE_NAME)
        .withCommonAttributes(commonAttributes);
    writeRecordsRequest.setRecords(records);

    try {
      WriteRecordsResult writeRecordsResult =
amazonTimestreamWrite.writeRecords(writeRecordsRequest);
      System.out.println("writeRecordsWithCommonAttributes Status: " +
writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("RejectedRecords: " + e);
      for (RejectedRecord rejectedRecord : e.getRejectedRecords()) {
        System.out.println("Rejected Index " + rejectedRecord.getRecordIndex() + ":
"
            + rejectedRecord.getReason());
      }
      System.out.println("Other records were written successfully. ");
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

Java v2

```
  public void writeRecordsWithCommonAttributes() {
    System.out.println("Writing records with extracting common attributes");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region = Dimension.builder().name("region").value("us-
east-1").build();
    final Dimension az = Dimension.builder().name("az").value("az1").build();
    final Dimension hostname =
 Dimension.builder().name("hostname").value("host1").build();

    dimensions.add(region);
    dimensions.add(az);
    dimensions.add(hostname);

    Record commonAttributes = Record.builder()
        .dimensions(dimensions)
        .measureValueType(MeasureValueType.DOUBLE)
        .time(String.valueOf(time)).build();

    Record cpuUtilization = Record.builder()
        .measureName("cpu_utilization")
        .measureValue("13.5").build();
    Record memoryUtilization = Record.builder()
        .measureName("memory_utilization")
        .measureValue("40").build();

    records.add(cpuUtilization);
    records.add(memoryUtilization);

    WriteRecordsRequest writeRecordsRequest = WriteRecordsRequest.builder()
        .databaseName(DATABASE_NAME)
        .tableName(TABLE_NAME)
        .commonAttributes(commonAttributes)
        .records(records).build();

    try {
      WriteRecordsResponse writeRecordsResponse =
 timestreamWriteClient.writeRecords(writeRecordsRequest);
```

```
      System.out.println("writeRecordsWithCommonAttributes Status: " +
 writeRecordsResponse.sdkHttpResponse().statusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("RejectedRecords: " + e);
      for (RejectedRecord rejectedRecord : e.rejectedRecords()) {
        System.out.println("Rejected Index " + rejectedRecord.recordIndex() + ": "
            + rejectedRecord.reason());
      }
      System.out.println("Other records were written successfully. ");
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

Go

```
now = time.Now()
currentTimeInSeconds = now.Unix()
writeRecordsCommonAttributesInput := &timestreamwrite.WriteRecordsInput{
 DatabaseName: aws.String(*databaseName),
 TableName:  aws.String(*tableName),
 CommonAttributes: &timestreamwrite.Record{
  Dimensions: []*timestreamwrite.Dimension{
   &timestreamwrite.Dimension{
    Name:  aws.String("region"),
    Value: aws.String("us-east-1"),
   },
   &timestreamwrite.Dimension{
    Name:  aws.String("az"),
    Value: aws.String("az1"),
   },
   &timestreamwrite.Dimension{
    Name:  aws.String("hostname"),
    Value: aws.String("host1"),
   },
  },
  MeasureValueType: aws.String("DOUBLE"),
  Time:       aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
  TimeUnit:     aws.String("SECONDS"),
 },
 Records: []*timestreamwrite.Record{
  &timestreamwrite.Record{
   MeasureName:  aws.String("cpu_utilization"),
```

```
    MeasureValue: aws.String("13.5"),
   },
   &timestreamwrite.Record{
    MeasureName:  aws.String("memory_utilization"),
    MeasureValue: aws.String("40"),
   },
  },
}

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesInput)

if err != nil {
 fmt.Println("Error:")
 fmt.Println(err)
} else {
 fmt.Println("Ingest records is successful")
}
```

Python

```
  def write_records_with_common_attributes(self):
    print("Writing records extracting common attributes")
    current_time = self._current_milli_time()

    dimensions = [
      {'Name': 'region', 'Value': 'us-east-1'},
      {'Name': 'az', 'Value': 'az1'},
      {'Name': 'hostname', 'Value': 'host1'}
    ]

    common_attributes = {
      'Dimensions': dimensions,
      'MeasureValueType': 'DOUBLE',
      'Time': current_time
    }

    cpu_utilization = {
      'MeasureName': 'cpu_utilization',
      'MeasureValue': '13.5'
    }

    memory_utilization = {
      'MeasureName': 'memory_utilization',
```

```
      'MeasureValue': '40'
    }

    records = [cpu_utilization, memory_utilization]

    try:
      result = self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                        Records=records, CommonAttributes=common_attributes)
      print("WriteRecords Status: [%s]" % result['ResponseMetadata']
['HTTPStatusCode'])
    except self.client.exceptions.RejectedRecordsException as err:
      self._print_rejected_records_exceptions(err)
    except Exception as err:
      print("Error:", err)

  @staticmethod
  def _print_rejected_records_exceptions(err):
    print("RejectedRecords: ", err)
    for rr in err.response["RejectedRecords"]:
      print("Rejected Index " + str(rr["RecordIndex"]) + ": " + rr["Reason"])
      if "ExistingVersion" in rr:
        print("Rejected record existing version: ", rr["ExistingVersion"])

  @staticmethod
  def _current_milli_time():
    return str(int(round(time.time() * 1000)))
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function writeRecordsWithCommonAttributes() {
  console.log("Writing records with common attributes");
  const currentTime = Date.now().toString(); // Unix time in milliseconds

  const dimensions = [
    {'Name': 'region', 'Value': 'us-east-1'},
    {'Name': 'az', 'Value': 'az1'},
    {'Name': 'hostname', 'Value': 'host1'}
  ];
```

```
  const commonAttributes = {
    'Dimensions': dimensions,
    'MeasureValueType': 'DOUBLE',
    'Time': currentTime.toString()
  };

  const cpuUtilization = {
    'MeasureName': 'cpu_utilization',
    'MeasureValue': '13.5'
  };

  const memoryUtilization = {
    'MeasureName': 'memory_utilization',
    'MeasureValue': '40'
  };

  const records = [cpuUtilization, memoryUtilization];

  const params = {
    DatabaseName: constants.DATABASE_NAME,
    TableName: constants.TABLE_NAME,
    Records: records,
    CommonAttributes: commonAttributes
  };

  const request = writeClient.writeRecords(params);

  await request.promise().then(
    (data) => {
      console.log("Write records successful");
    },
    (err) => {
      console.log("Error writing records:", err);
      if (err.code === 'RejectedRecordsException') {
        const responsePayload =
 JSON.parse(request.response.httpResponse.body.toString());
        console.log("RejectedRecords: ", responsePayload.RejectedRecords);
        console.log("Other records were written successfully. ");
      }
    }
  );
}
```

.NET

```
public async Task WriteRecordsWithCommonAttributes()
{
  Console.WriteLine("Writing records with common attributes");

  DateTimeOffset now = DateTimeOffset.UtcNow;
  string currentTimeString = (now.ToUnixTimeMilliseconds()).ToString();

  List<Dimension> dimensions = new List<Dimension>{
    new Dimension { Name = "region", Value = "us-east-1" },
    new Dimension { Name = "az", Value = "az1" },
    new Dimension { Name = "hostname", Value = "host1" }
  };

  var commonAttributes = new Record
  {
    Dimensions = dimensions,
    MeasureValueType = MeasureValueType.DOUBLE,
    Time = currentTimeString
  };

  var cpuUtilization = new Record
  {
    MeasureName = "cpu_utilization",
    MeasureValue = "13.6"
  };

  var memoryUtilization = new Record
  {
    MeasureName = "memory_utilization",
    MeasureValue = "40"
  };


  List<Record> records = new List<Record>();
  records.Add(cpuUtilization);
  records.Add(memoryUtilization);

  try
  {
    var writeRecordsRequest = new WriteRecordsRequest
    {
      DatabaseName = Constants.DATABASE_NAME,
```

```
        TableName = Constants.TABLE_NAME,
        Records = records,
        CommonAttributes = commonAttributes
      };
      WriteRecordsResponse response = await
writeClient.WriteRecordsAsync(writeRecordsRequest);
      Console.WriteLine($"Write records status code:
{response.HttpStatusCode.ToString()}");
    }
    catch (RejectedRecordsException e) {
      Console.WriteLine("RejectedRecordsException:" + e.ToString());
      foreach (RejectedRecord rr in e.RejectedRecords) {
        Console.WriteLine("RecordIndex " + rr.RecordIndex + " : " + rr.Reason);
      }
      Console.WriteLine("Other records were written successfully. ");
    }
    catch (Exception e)
    {
      Console.WriteLine("Write records failure:" + e.ToString());
    }
  }
```

Upserting records

While the default writes in Amazon Timestream follow the *first writer wins* semantics, where data is stored as append only and duplicate records are rejected, there are applications that require the ability to write data into Amazon Timestream using the *last writer wins* semantics, where the record with the highest version is stored in the system. There are also applications that require the ability to update existing records. To address these scenarios, Amazon Timestream provides the ability to *upsert* data. Upsert is an operation that inserts a record in to the system when the record does not exist or updates the record, when one exists.

You can upsert records by including the `Version` in record definition while sending a `WriteRecords` request. Amazon Timestream will store the record with the record with highest `Version`. The code sample below shows how you can upsert data:

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
  public void writeRecordsWithUpsert() {
    System.out.println("Writing records with upsert");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();
    // To achieve upsert (last writer wins) semantic, one example is to use current
 time as the version if you are writing directly from the data source
    long version = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region = new Dimension().withName("region").withValue("us-
east-1");
    final Dimension az = new Dimension().withName("az").withValue("az1");
    final Dimension hostname = new
 Dimension().withName("hostname").withValue("host1");

    dimensions.add(region);
    dimensions.add(az);
    dimensions.add(hostname);

    Record commonAttributes = new Record()
        .withDimensions(dimensions)
        .withMeasureValueType(MeasureValueType.DOUBLE)
        .withTime(String.valueOf(time))
        .withVersion(version);

    Record cpuUtilization = new Record()
        .withMeasureName("cpu_utilization")
        .withMeasureValue("13.5");
    Record memoryUtilization = new Record()
        .withMeasureName("memory_utilization")
        .withMeasureValue("40");

    records.add(cpuUtilization);
    records.add(memoryUtilization);

    WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
        .withDatabaseName(DATABASE_NAME)
        .withTableName(TABLE_NAME)
        .withCommonAttributes(commonAttributes);
    writeRecordsRequest.setRecords(records);
```

```
   // write records for first time
   try {
     WriteRecordsResult writeRecordsResult =
amazonTimestreamWrite.writeRecords(writeRecordsRequest);
     System.out.println("WriteRecords Status for first time: " +
writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
   } catch (RejectedRecordsException e) {
     printRejectedRecordsException(e);
   } catch (Exception e) {
     System.out.println("Error: " + e);
   }

   // Successfully retry same writeRecordsRequest with same records and versions,
because writeRecords API is idempotent.
   try {
     WriteRecordsResult writeRecordsResult =
amazonTimestreamWrite.writeRecords(writeRecordsRequest);
     System.out.println("WriteRecords Status for retry: " +
writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
   } catch (RejectedRecordsException e) {
     printRejectedRecordsException(e);
   } catch (Exception e) {
     System.out.println("Error: " + e);
   }

   // upsert with lower version, this would fail because a higher version is
required to update the measure value.
   version -= 1;
   commonAttributes.setVersion(version);

   cpuUtilization.setMeasureValue("14.5");
   memoryUtilization.setMeasureValue("50");

   List<Record> upsertedRecords = new ArrayList<>();
   upsertedRecords.add(cpuUtilization);
   upsertedRecords.add(memoryUtilization);

   WriteRecordsRequest writeRecordsUpsertRequest = new WriteRecordsRequest()
       .withDatabaseName(DATABASE_NAME)
       .withTableName(TABLE_NAME)
       .withCommonAttributes(commonAttributes);
   writeRecordsUpsertRequest.setRecords(upsertedRecords);

   try {
```

```
      WriteRecordsResult writeRecordsUpsertResult =
amazonTimestreamWrite.writeRecords(writeRecordsUpsertRequest);
      System.out.println("WriteRecords Status for upsert with lower version: " +
writeRecordsUpsertResult.getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("WriteRecords Status for upsert with lower version: ");
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }

    // upsert with higher version as new data in generated
    version = System.currentTimeMillis();
    commonAttributes.setVersion(version);

    writeRecordsUpsertRequest = new WriteRecordsRequest()
        .withDatabaseName(DATABASE_NAME)
        .withTableName(TABLE_NAME)
        .withCommonAttributes(commonAttributes);
    writeRecordsUpsertRequest.setRecords(upsertedRecords);

    try {
      WriteRecordsResult writeRecordsUpsertResult =
amazonTimestreamWrite.writeRecords(writeRecordsUpsertRequest);
      System.out.println("WriteRecords Status for upsert with higher version: " +
writeRecordsUpsertResult.getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

Java v2

```
  public void writeRecordsWithUpsert() {
    System.out.println("Writing records with upsert");
    // Specify repeated values for all records
    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();
    // To achieve upsert (last writer wins) semantic, one example is to use current
time as the version if you are writing directly from the data source
    long version = System.currentTimeMillis();
```

```
    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region = Dimension.builder().name("region").value("us-
east-1").build();
    final Dimension az = Dimension.builder().name("az").value("az1").build();
    final Dimension hostname =
 Dimension.builder().name("hostname").value("host1").build();

    dimensions.add(region);
    dimensions.add(az);
    dimensions.add(hostname);

    Record commonAttributes = Record.builder()
        .dimensions(dimensions)
        .measureValueType(MeasureValueType.DOUBLE)
        .time(String.valueOf(time))
        .version(version)
        .build();

    Record cpuUtilization = Record.builder()
        .measureName("cpu_utilization")
        .measureValue("13.5").build();
    Record memoryUtilization = Record.builder()
        .measureName("memory_utilization")
        .measureValue("40").build();

    records.add(cpuUtilization);
    records.add(memoryUtilization);

    WriteRecordsRequest writeRecordsRequest = WriteRecordsRequest.builder()
        .databaseName(DATABASE_NAME)
        .tableName(TABLE_NAME)
        .commonAttributes(commonAttributes)
        .records(records).build();

    // write records for first time
    try {
      WriteRecordsResponse writeRecordsResponse =
 timestreamWriteClient.writeRecords(writeRecordsRequest);
      System.out.println("WriteRecords Status for first time: " +
 writeRecordsResponse.sdkHttpResponse().statusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
```

```
      System.out.println("Error: " + e);
    }

    // Successfully retry same writeRecordsRequest with same records and versions,
because writeRecords API is idempotent.
    try {
      WriteRecordsResponse writeRecordsResponse =
timestreamWriteClient.writeRecords(writeRecordsRequest);
      System.out.println("WriteRecords Status for retry: " +
writeRecordsResponse.sdkHttpResponse().statusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }

    // upsert with lower version, this would fail because a higher version is
required to update the measure value.
    version -= 1;
    commonAttributes = Record.builder()
        .dimensions(dimensions)
        .measureValueType(MeasureValueType.DOUBLE)
        .time(String.valueOf(time))
        .version(version)
        .build();

    cpuUtilization = Record.builder()
        .measureName("cpu_utilization")
        .measureValue("14.5").build();
    memoryUtilization = Record.builder()
        .measureName("memory_utilization")
        .measureValue("50").build();

    List<Record> upsertedRecords = new ArrayList<>();
    upsertedRecords.add(cpuUtilization);
    upsertedRecords.add(memoryUtilization);

    WriteRecordsRequest writeRecordsUpsertRequest = WriteRecordsRequest.builder()
        .databaseName(DATABASE_NAME)
        .tableName(TABLE_NAME)
        .commonAttributes(commonAttributes)
        .records(upsertedRecords).build();

    try {
```

```
      WriteRecordsResponse writeRecordsResponse =
 timestreamWriteClient.writeRecords(writeRecordsUpsertRequest);
      System.out.println("WriteRecords Status for upsert with lower version: " +
 writeRecordsResponse.sdkHttpResponse().statusCode());
    } catch (RejectedRecordsException e) {
      System.out.println("WriteRecords Status for upsert with lower version: ");
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }

    // upsert with higher version as new data in generated
    version = System.currentTimeMillis();
    commonAttributes = Record.builder()
        .dimensions(dimensions)
        .measureValueType(MeasureValueType.DOUBLE)
        .time(String.valueOf(time))
        .version(version)
        .build();

    writeRecordsUpsertRequest = WriteRecordsRequest.builder()
        .databaseName(DATABASE_NAME)
        .tableName(TABLE_NAME)
        .commonAttributes(commonAttributes)
        .records(upsertedRecords).build();

    try {
      WriteRecordsResponse writeRecordsUpsertResponse =
 timestreamWriteClient.writeRecords(writeRecordsUpsertRequest);
      System.out.println("WriteRecords Status for upsert with higher version: " +
 writeRecordsUpsertResponse.sdkHttpResponse().statusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

Go

```
// Below code will ingest and upsert cpu_utilization and memory_utilization metric
 for a host on
// region=us-east-1, az=az1, and hostname=host1
```

```
fmt.Println("Ingesting records and set version as currentTimeInMills, hit enter to
 continue")
reader.ReadString('\n')

// Get current time in seconds.
now = time.Now()
currentTimeInSeconds = now.Unix()
// To achieve upsert (last writer wins) semantic, one example is to use current time
 as the version if you are writing directly from the data source
version := time.Now().Round(time.Millisecond).UnixNano() / 1e6   // set version as
 currentTimeInMills

writeRecordsCommonAttributesUpsertInput := &timestreamwrite.WriteRecordsInput{
 DatabaseName: aws.String(*databaseName),
 TableName:  aws.String(*tableName),
 CommonAttributes: &timestreamwrite.Record{
  Dimensions: []*timestreamwrite.Dimension{
   &timestreamwrite.Dimension{
    Name:  aws.String("region"),
    Value: aws.String("us-east-1"),
   },
   &timestreamwrite.Dimension{
    Name:  aws.String("az"),
    Value: aws.String("az1"),
   },
   &timestreamwrite.Dimension{
    Name:  aws.String("hostname"),
    Value: aws.String("host1"),
   },
  },
  MeasureValueType: aws.String("DOUBLE"),
  Time:       aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
  TimeUnit:  aws.String("SECONDS"),
  Version:      &version,
 },
 Records: []*timestreamwrite.Record{
  &timestreamwrite.Record{
   MeasureName:  aws.String("cpu_utilization"),
   MeasureValue: aws.String("13.5"),
  },
  &timestreamwrite.Record{
   MeasureName:  aws.String("memory_utilization"),
   MeasureValue: aws.String("40"),
  },
```

```
 },
}

// write records for first time
_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

if err != nil {
 fmt.Println("Error:")
 fmt.Println(err)
} else {
 fmt.Println("Frist-time write records is successful")
}

fmt.Println("Retry same writeRecordsRequest with same records and versions. Because
 writeRecords API is idempotent, this will success. hit enter to continue")
reader.ReadString('\n')

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

if err != nil {
 fmt.Println("Error:")
 fmt.Println(err)
} else {
 fmt.Println("Retry write records for same request is successful")
}

fmt.Println("Upsert with lower version, this would fail because a higher version is
 required to update the measure value. hit enter to continue")
reader.ReadString('\n')
version -= 1
writeRecordsCommonAttributesUpsertInput.CommonAttributes.Version = &version

updated_cpu_utilization := &timestreamwrite.Record{
 MeasureName:    aws.String("cpu_utilization"),
 MeasureValue:   aws.String("14.5"),
}
updated_memory_utilization := &timestreamwrite.Record{
 MeasureName:    aws.String("memory_utilization"),
 MeasureValue:   aws.String("50"),
}


writeRecordsCommonAttributesUpsertInput.Records = []*timestreamwrite.Record{
 updated_cpu_utilization,
```

```
 updated_memory_utilization,
}

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

if err != nil {
 fmt.Println("Error:")
 fmt.Println(err)
} else {
 fmt.Println("Write records with lower version is successful")
}

fmt.Println("Upsert with higher version as new data in generated, this would
 success. hit enter to continue")
reader.ReadString('\n')

version = time.Now().Round(time.Millisecond).UnixNano() / 1e6  // set version as
 currentTimeInMills
writeRecordsCommonAttributesUpsertInput.CommonAttributes.Version = &version

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

if err != nil {
 fmt.Println("Error:")
 fmt.Println(err)
} else {
 fmt.Println("Write records with higher version is successful")
}
```

Python

```
  def write_records_with_upsert(self):
    print("Writing records with upsert")
    current_time = self._current_milli_time()
    # To achieve upsert (last writer wins) semantic, one example is to use current
 time as the version if you are writing directly from the data source
    version = int(self._current_milli_time())

    dimensions = [
          {'Name': 'region', 'Value': 'us-east-1'},
          {'Name': 'az', 'Value': 'az1'},
          {'Name': 'hostname', 'Value': 'host1'}
        ]
```

```
    common_attributes = {
      'Dimensions': dimensions,
      'MeasureValueType': 'DOUBLE',
      'Time': current_time,
      'Version': version
    }

    cpu_utilization = {
      'MeasureName': 'cpu_utilization',
      'MeasureValue': '13.5'
    }

    memory_utilization = {
      'MeasureName': 'memory_utilization',
      'MeasureValue': '40'
    }

    records = [cpu_utilization, memory_utilization]

    # write records for first time
    try:
      result = self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                         Records=records, CommonAttributes=common_attributes)
      print("WriteRecords Status for first time: [%s]" % result['ResponseMetadata']
['HTTPStatusCode'])
    except self.client.exceptions.RejectedRecordsException as err:
      self._print_rejected_records_exceptions(err)
    except Exception as err:
      print("Error:", err)

    # Successfully retry same writeRecordsRequest with same records and versions,
 because writeRecords API is idempotent.
    try:
      result = self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                         Records=records, CommonAttributes=common_attributes)
      print("WriteRecords Status for retry: [%s]" % result['ResponseMetadata']
['HTTPStatusCode'])
    except self.client.exceptions.RejectedRecordsException as err:
      self._print_rejected_records_exceptions(err)
    except Exception as err:
      print("Error:", err)
```

```
    # upsert with lower version, this would fail because a higher version is
 required to update the measure value.
    version -= 1
    common_attributes["Version"] = version

    cpu_utilization["MeasureValue"] = '14.5'
    memory_utilization["MeasureValue"] = '50'

    upsertedRecords = [cpu_utilization, memory_utilization]

    try:
      upsertedResult =
 self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                            Records=upsertedRecords,
 CommonAttributes=common_attributes)
      print("WriteRecords Status for upsert with lower version: [%s]" %
 upsertedResult['ResponseMetadata']['HTTPStatusCode'])
    except self.client.exceptions.RejectedRecordsException as err:
      self._print_rejected_records_exceptions(err)
    except Exception as err:
      print("Error:", err)


    # upsert with higher version as new data is generated
    version = int(self._current_milli_time())
    common_attributes["Version"] = version

    try:
      upsertedResult =
 self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
                            Records=upsertedRecords,
 CommonAttributes=common_attributes)
      print("WriteRecords Upsert Status: [%s]" % upsertedResult['ResponseMetadata']
['HTTPStatusCode'])
    except self.client.exceptions.RejectedRecordsException as err:
      self._print_rejected_records_exceptions(err)
    except Exception as err:
      print("Error:", err)

  @staticmethod
  def _current_milli_time():
```

```
    return str(int(round(time.time() * 1000)))
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function writeRecordsWithUpsert() {
  console.log("Writing records with upsert");
  const currentTime = Date.now().toString(); // Unix time in milliseconds
  // To achieve upsert (last writer wins) semantic, one example is to use current
 time as the version if you are writing directly from the data source
  let version = Date.now();

  const dimensions = [
    {'Name': 'region', 'Value': 'us-east-1'},
    {'Name': 'az', 'Value': 'az1'},
    {'Name': 'hostname', 'Value': 'host1'}
  ];

  const commonAttributes = {
    'Dimensions': dimensions,
    'MeasureValueType': 'DOUBLE',
    'Time': currentTime.toString(),
    'Version': version
  };

  const cpuUtilization = {
    'MeasureName': 'cpu_utilization',
    'MeasureValue': '13.5'
  };

  const memoryUtilization = {
    'MeasureName': 'memory_utilization',
    'MeasureValue': '40'
  };

  const records = [cpuUtilization, memoryUtilization];

  const params = {
    DatabaseName: constants.DATABASE_NAME,
    TableName: constants.TABLE_NAME,
    Records: records,
```

```
    CommonAttributes: commonAttributes
  };

  const request = writeClient.writeRecords(params);

  // write records for first time
  await request.promise().then(
    (data) => {
      console.log("Write records successful for first time.");
    },
    (err) => {
      console.log("Error writing records:", err);
      if (err.code === 'RejectedRecordsException') {
        printRejectedRecordsException(request);
      }
    }
  );

  // Successfully retry same writeRecordsRequest with same records and versions,
because writeRecords API is idempotent.
  await request.promise().then(
    (data) => {
      console.log("Write records successful for retry.");
    },
    (err) => {
      console.log("Error writing records:", err);
      if (err.code === 'RejectedRecordsException') {
        printRejectedRecordsException(request);
      }
    }
  );

  // upsert with lower version, this would fail because a higher version is required
to update the measure value.
  version--;

  const commonAttributesWithLowerVersion = {
    'Dimensions': dimensions,
    'MeasureValueType': 'DOUBLE',
    'Time': currentTime.toString(),
    'Version': version
  };

  const updatedCpuUtilization = {
```

```
    'MeasureName': 'cpu_utilization',
    'MeasureValue': '14.5'
  };

  const updatedMemoryUtilization = {
    'MeasureName': 'memory_utilization',
    'MeasureValue': '50'
  };

  const upsertedRecords = [updatedCpuUtilization, updatedMemoryUtilization];

  const upsertedParamsWithLowerVersion = {
    DatabaseName: constants.DATABASE_NAME,
    TableName: constants.TABLE_NAME,
    Records: upsertedRecords,
    CommonAttributes: commonAttributesWithLowerVersion
  };

  const upsertRequestWithLowerVersion =
writeClient.writeRecords(upsertedParamsWithLowerVersion);

  await upsertRequestWithLowerVersion.promise().then(
    (data) => {
      console.log("Write records for upsert with lower version successful");
    },
    (err) => {
      console.log("Error writing records:", err);
      if (err.code === 'RejectedRecordsException') {
        printRejectedRecordsException(upsertRequestWithLowerVersion);
      }
    }
  );

  // upsert with higher version as new data in generated
  version = Date.now();

  const commonAttributesWithHigherVersion = {
    'Dimensions': dimensions,
    'MeasureValueType': 'DOUBLE',
    'Time': currentTime.toString(),
    'Version': version
  };

  const upsertedParamsWithHigherVerion = {
```

```
    DatabaseName: constants.DATABASE_NAME,
    TableName: constants.TABLE_NAME,
    Records: upsertedRecords,
    CommonAttributes: commonAttributesWithHigherVersion
  };

  const upsertRequestWithHigherVersion =
 writeClient.writeRecords(upsertedParamsWithHigherVerion);

  await upsertRequestWithHigherVersion.promise().then(
    (data) => {
      console.log("Write records upsert successful with higher version");
    },
    (err) => {
      console.log("Error writing records:", err);
      if (err.code === 'RejectedRecordsException') {
        printRejectedRecordsException(upsertedParamsWithHigherVerion);
      }
    }
  );

}
```

.NET

```
  public async Task WriteRecordsWithUpsert()
  {
    Console.WriteLine("Writing records with upsert");

    DateTimeOffset now = DateTimeOffset.UtcNow;
    string currentTimeString = (now.ToUnixTimeMilliseconds()).ToString();
    // To achieve upsert (last writer wins) semantic, one example is to use current
 time as the version if you are writing directly from the data source
    long version = now.ToUnixTimeMilliseconds();

    List<Dimension> dimensions = new List<Dimension>{
      new Dimension { Name = "region", Value = "us-east-1" },
      new Dimension { Name = "az", Value = "az1" },
      new Dimension { Name = "hostname", Value = "host1" }
    };

    var commonAttributes = new Record
    {
```

```
      Dimensions = dimensions,
      MeasureValueType = MeasureValueType.DOUBLE,
      Time = currentTimeString,
      Version = version
    };

    var cpuUtilization = new Record
    {
      MeasureName = "cpu_utilization",
      MeasureValue = "13.6"
    };

    var memoryUtilization = new Record
    {
      MeasureName = "memory_utilization",
      MeasureValue = "40"
    };


    List<Record> records = new List<Record>();
    records.Add(cpuUtilization);
    records.Add(memoryUtilization);

    // write records for first time
    try
    {
      var writeRecordsRequest = new WriteRecordsRequest
      {
        DatabaseName = Constants.DATABASE_NAME,
        TableName = Constants.TABLE_NAME,
        Records = records,
        CommonAttributes = commonAttributes
      };
      WriteRecordsResponse response = await
 writeClient.WriteRecordsAsync(writeRecordsRequest);
      Console.WriteLine($"WriteRecords Status for first time:
 {response.HttpStatusCode.ToString()}");
    }
    catch (RejectedRecordsException e) {
      PrintRejectedRecordsException(e);
    }
    catch (Exception e)
    {
      Console.WriteLine("Write records failure:" + e.ToString());
```

```
   }

   // Successfully retry same writeRecordsRequest with same records and versions,
because writeRecords API is idempotent.
   try
   {
     var writeRecordsRequest = new WriteRecordsRequest
     {
       DatabaseName = Constants.DATABASE_NAME,
       TableName = Constants.TABLE_NAME,
       Records = records,
       CommonAttributes = commonAttributes
     };
     WriteRecordsResponse response = await
writeClient.WriteRecordsAsync(writeRecordsRequest);
     Console.WriteLine($"WriteRecords Status for retry:
{response.HttpStatusCode.ToString()}");
   }
   catch (RejectedRecordsException e) {
     PrintRejectedRecordsException(e);
   }
   catch (Exception e)
   {
     Console.WriteLine("Write records failure:" + e.ToString());
   }

   // upsert with lower version, this would fail because a higher version is
required to update the measure value.
   version--;
   Type recordType = typeof(Record);
   recordType.GetProperty("Version").SetValue(commonAttributes, version);
   recordType.GetProperty("MeasureValue").SetValue(cpuUtilization, "14.6");
   recordType.GetProperty("MeasureValue").SetValue(memoryUtilization, "50");

   List<Record> upsertedRecords = new List<Record> {
     cpuUtilization,
     memoryUtilization
   };

   try
   {
     var writeRecordsUpsertRequest = new WriteRecordsRequest
     {
       DatabaseName = Constants.DATABASE_NAME,
```

```
        TableName = Constants.TABLE_NAME,
        Records = upsertedRecords,
        CommonAttributes = commonAttributes
      };
      WriteRecordsResponse upsertResponse = await
writeClient.WriteRecordsAsync(writeRecordsUpsertRequest);
      Console.WriteLine($"WriteRecords Status for upsert with lower version:
{upsertResponse.HttpStatusCode.ToString()}");
    }
    catch (RejectedRecordsException e) {
      PrintRejectedRecordsException(e);
    }
    catch (Exception e)
    {
      Console.WriteLine("Write records failure:" + e.ToString());
    }

    // upsert with higher version as new data in generated
    now = DateTimeOffset.UtcNow;
    version = now.ToUnixTimeMilliseconds();
    recordType.GetProperty("Version").SetValue(commonAttributes, version);

    try
    {
      var writeRecordsUpsertRequest = new WriteRecordsRequest
      {
        DatabaseName = Constants.DATABASE_NAME,
        TableName = Constants.TABLE_NAME,
        Records = upsertedRecords,
        CommonAttributes = commonAttributes
      };
      WriteRecordsResponse upsertResponse = await
writeClient.WriteRecordsAsync(writeRecordsUpsertRequest);
      Console.WriteLine($"WriteRecords Status for upsert with higher version:
{upsertResponse.HttpStatusCode.ToString()}");
    }
    catch (RejectedRecordsException e) {
      PrintRejectedRecordsException(e);
    }
    catch (Exception e)
    {
      Console.WriteLine("Write records failure:" + e.ToString());
    }
```

```
  }
```

Multi-measure attribute example

This example illustrates writing multi-mearure attributes. Multi-measure attributes are useful when a device or an application you are tracking emits multiple metrics or events at the same timestamp..

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
package com.amazonaws.services.timestream;

import static com.amazonaws.services.timestream.Main.DATABASE_NAME;
import static com.amazonaws.services.timestream.Main.REGION;
import static com.amazonaws.services.timestream.Main.TABLE_NAME;

import java.util.ArrayList;
import java.util.List;

import com.amazonaws.services.timestreamwrite.AmazonTimestreamWrite;
import com.amazonaws.services.timestreamwrite.model.Dimension;
import com.amazonaws.services.timestreamwrite.model.MeasureValue;
import com.amazonaws.services.timestreamwrite.model.MeasureValueType;
import com.amazonaws.services.timestreamwrite.model.Record;
import com.amazonaws.services.timestreamwrite.model.RejectedRecordsException;
import com.amazonaws.services.timestreamwrite.model.WriteRecordsRequest;
import com.amazonaws.services.timestreamwrite.model.WriteRecordsResult;


public class multimeasureAttributeExample {
  AmazonTimestreamWrite timestreamWriteClient;

  public multimeasureAttributeExample(AmazonTimestreamWrite client) {
    this.timestreamWriteClient = client;
  }
```

```
 public void writeRecordsMultiMeasureValueSingleRecord() {
   System.out.println("Writing records with multi value attributes");

   List<Record> records = new ArrayList<>();
   final long time = System.currentTimeMillis();
   long version = System.currentTimeMillis();

   List<Dimension> dimensions = new ArrayList<>();
   final Dimension region = new Dimension().withName("region").withValue(REGION);
   final Dimension az = new Dimension().withName("az").withValue("az1");
   final Dimension hostname = new
Dimension().withName("hostname").withValue("host1");

   dimensions.add(region);
   dimensions.add(az);
   dimensions.add(hostname);

   Record commonAttributes = new Record()
       .withDimensions(dimensions)
       .withTime(String.valueOf(time))
       .withVersion(version);

   MeasureValue cpuUtilization = new MeasureValue()
       .withName("cpu_utilization")
       .withType(MeasureValueType.DOUBLE)
       .withValue("13.5");
   MeasureValue memoryUtilization = new MeasureValue()
       .withName("memory_utilization")
       .withType(MeasureValueType.DOUBLE)
       .withValue("40");
   Record computationalResources = new Record()
       .withMeasureName("cpu_memory")
       .withMeasureValues(cpuUtilization, memoryUtilization)
       .withMeasureValueType(MeasureValueType.MULTI);

   records.add(computationalResources);

   WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
       .withDatabaseName(DATABASE_NAME)
       .withTableName(TABLE_NAME)
       .withCommonAttributes(commonAttributes)
       .withRecords(records);
```

```
    // write records for first time
    try {
      WriteRecordsResult writeRecordResult =
timestreamWriteClient.writeRecords(writeRecordsRequest);
      System.out.println(
          "WriteRecords Status for multi value attributes: " + writeRecordResult
              .getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }

  public void writeRecordsMultiMeasureValueMultipleRecords() {
    System.out.println(
        "Writing records with multi value attributes mixture type");

    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();
    long version = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region = new Dimension().withName("region").withValue(REGION);
    final Dimension az = new Dimension().withName("az").withValue("az1");
    final Dimension hostname = new
Dimension().withName("hostname").withValue("host1");

    dimensions.add(region);
    dimensions.add(az);
    dimensions.add(hostname);

    Record commonAttributes = new Record()
        .withDimensions(dimensions)
        .withTime(String.valueOf(time))
        .withVersion(version);

    MeasureValue cpuUtilization = new MeasureValue()
        .withName("cpu_utilization")
        .withType(MeasureValueType.DOUBLE)
        .withValue("13");
    MeasureValue memoryUtilization =new MeasureValue()
        .withName("memory_utilization")
        .withType(MeasureValueType.DOUBLE)
```

```
        .withValue("40");
    MeasureValue activeCores = new MeasureValue()
        .withName("active_cores")
        .withType(MeasureValueType.BIGINT)
        .withValue("4");


    Record computationalResources = new Record()
        .withMeasureName("computational_utilization")
        .withMeasureValues(cpuUtilization, memoryUtilization, activeCores)
        .withMeasureValueType(MeasureValueType.MULTI);

    records.add(computationalResources);

    WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
        .withDatabaseName(DATABASE_NAME)
        .withTableName(TABLE_NAME)
        .withCommonAttributes(commonAttributes)
        .withRecords(records);

    // write records for first time
    try {
      WriteRecordsResult writeRecordResult =
 timestreamWriteClient.writeRecords(writeRecordsRequest);
      System.out.println(
          "WriteRecords Status for multi value attributes: " + writeRecordResult
              .getSdkHttpMetadata().getHttpStatusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }

  private void printRejectedRecordsException(RejectedRecordsException e) {
    System.out.println("RejectedRecords: " + e);
    e.getRejectedRecords().forEach(System.out::println);
  }
}
```

Java v2

```
package com.amazonaws.services.timestream;
```

```
import java.util.ArrayList;
import java.util.List;

import software.amazon.awssdk.services.timestreamwrite.TimestreamWriteClient;
import software.amazon.awssdk.services.timestreamwrite.model.Dimension;
import software.amazon.awssdk.services.timestreamwrite.model.MeasureValue;
import software.amazon.awssdk.services.timestreamwrite.model.MeasureValueType;
import software.amazon.awssdk.services.timestreamwrite.model.Record;
import
 software.amazon.awssdk.services.timestreamwrite.model.RejectedRecordsException;
import software.amazon.awssdk.services.timestreamwrite.model.WriteRecordsRequest;
import software.amazon.awssdk.services.timestreamwrite.model.WriteRecordsResponse;

import static com.amazonaws.services.timestream.Main.DATABASE_NAME;
import static com.amazonaws.services.timestream.Main.TABLE_NAME;


public class multimeasureAttributeExample {

  TimestreamWriteClient timestreamWriteClient;

  public multimeasureAttributeExample(TimestreamWriteClient client) {
    this.timestreamWriteClient = client;
  }

  public void writeRecordsMultiMeasureValueSingleRecord() {
    System.out.println("Writing records with multi value attributes");

    List<Record> records = new ArrayList<>();
    final long time = System.currentTimeMillis();
    long version = System.currentTimeMillis();

    List<Dimension> dimensions = new ArrayList<>();
    final Dimension region =
        Dimension.builder().name("region").value("us-east-1").build();
    final Dimension az = Dimension.builder().name("az").value("az1").build();
    final Dimension hostname =
        Dimension.builder().name("hostname").value("host1").build();

    dimensions.add(region);
    dimensions.add(az);
    dimensions.add(hostname);
```

```
    Record commonAttributes = Record.builder()
        .dimensions(dimensions)
        .time(String.valueOf(time))
        .version(version)
        .build();

    MeasureValue cpuUtilization = MeasureValue.builder()
        .name("cpu_utilization")
        .type(MeasureValueType.DOUBLE)
        .value("13.5").build();
    MeasureValue memoryUtilization = MeasureValue.builder()
        .name("memory_utilization")
        .type(MeasureValueType.DOUBLE)
        .value("40").build();
    Record computationalResources = Record
        .builder()
        .measureName("cpu_memory")
        .measureValues(cpuUtilization, memoryUtilization)
        .measureValueType(MeasureValueType.MULTI)
        .build();

    records.add(computationalResources);

    WriteRecordsRequest writeRecordsRequest = WriteRecordsRequest.builder()
        .databaseName(DATABASE_NAME)
        .tableName(TABLE_NAME)
        .commonAttributes(commonAttributes)
        .records(records).build();

    // write records for first time
    try {
      WriteRecordsResponse writeRecordsResponse =
timestreamWriteClient.writeRecords(writeRecordsRequest);
      System.out.println(
          "WriteRecords Status for multi value attributes: " + writeRecordsResponse
              .sdkHttpResponse()
              .statusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
```

```
public void writeRecordsMultiMeasureValueMultipleRecords() {
  System.out.println(
      "Writing records with multi value attributes mixture type");

  List<Record> records = new ArrayList<>();
  final long time = System.currentTimeMillis();
  long version = System.currentTimeMillis();

  List<Dimension> dimensions = new ArrayList<>();
  final Dimension region =
      Dimension.builder().name("region").value("us-east-1").build();
  final Dimension az = Dimension.builder().name("az").value("az1").build();
  final Dimension hostname =
      Dimension.builder().name("hostname").value("host1").build();

  dimensions.add(region);
  dimensions.add(az);
  dimensions.add(hostname);

  Record commonAttributes = Record.builder()
      .dimensions(dimensions)
      .time(String.valueOf(time))
      .version(version)
      .build();

  MeasureValue cpuUtilization = MeasureValue.builder()
      .name("cpu_utilization")
      .type(MeasureValueType.DOUBLE)
      .value("13.5").build();
  MeasureValue memoryUtilization = MeasureValue.builder()
      .name("memory_utilization")
      .type(MeasureValueType.DOUBLE)
      .value("40").build();
  MeasureValue activeCores = MeasureValue.builder()
      .name("active_cores")
      .type(MeasureValueType.BIGINT)
      .value("4").build();


  Record computationalResources = Record
      .builder()
      .measureName("computational_utilization")
      .measureValues(cpuUtilization, memoryUtilization, activeCores)
      .measureValueType(MeasureValueType.MULTI)
```

```
        .build();

    records.add(computationalResources);

    WriteRecordsRequest writeRecordsRequest = WriteRecordsRequest.builder()
        .databaseName(DATABASE_NAME)
        .tableName(TABLE_NAME)
        .commonAttributes(commonAttributes)
        .records(records).build();

    // write records for first time
    try {
      WriteRecordsResponse writeRecordsResponse =
 timestreamWriteClient.writeRecords(writeRecordsRequest);
      System.out.println(
          "WriteRecords Status for multi value attributes: " + writeRecordsResponse
              .sdkHttpResponse()
              .statusCode());
    } catch (RejectedRecordsException e) {
      printRejectedRecordsException(e);
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }

  private void printRejectedRecordsException(RejectedRecordsException e) {
    System.out.println("RejectedRecords: " + e);
    e.rejectedRecords().forEach(System.out::println);
  }
}
```

Go

```
  now := time.Now()
  currentTimeInSeconds := now.Unix()
  writeRecordsInput := &timestreamwrite.WriteRecordsInput{
    DatabaseName: aws.String(*databaseName),
    TableName:  aws.String(*tableName),
    Records: []*timestreamwrite.Record{
    &timestreamwrite.Record{
      Dimensions: []*timestreamwrite.Dimension{
      &timestreamwrite.Dimension{
        Name:  aws.String("region"),
```

```
        Value: aws.String("us-east-1"),
      },
      &timestreamwrite.Dimension{
        Name:  aws.String("az"),
        Value: aws.String("az1"),
      },
      &timestreamwrite.Dimension{
        Name:  aws.String("hostname"),
        Value: aws.String("host1"),
      },
      },
      MeasureName:  aws.String("metrics"),
      MeasureValueType: aws.String("MULTI"),
      Time:     aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
      TimeUnit:  aws.String("SECONDS"),
      MeasureValues: []*timestreamwrite.MeasureValue{
      &timestreamwrite.MeasureValue{
        Name:  aws.String("cpu_utilization"),
        Value: aws.String("13.5"),
        Type:  aws.String("DOUBLE"),
      },
      &timestreamwrite.MeasureValue{
        Name:  aws.String("memory_utilization"),
        Value: aws.String("40"),
        Type:  aws.String("DOUBLE"),
      },
      },
    },
    },
  }

  _, err = writeSvc.WriteRecords(writeRecordsInput)

  if err != nil {
    fmt.Println("Error:")
    fmt.Println(err)
  } else {
    fmt.Println("Write records is successful")
  }
```

Python

```
import time
```

```
import boto3
import psutil
import os

from botocore.config import Config

DATABASE_NAME = os.environ['DATABASE_NAME']
TABLE_NAME = os.environ['TABLE_NAME']

COUNTRY = "UK"
CITY = "London"
HOSTNAME = "MyHostname" # You can make it dynamic using socket.gethostname()

INTERVAL = 1 # Seconds

def prepare_common_attributes():
  common_attributes = {
    'Dimensions': [
      {'Name': 'country', 'Value': COUNTRY},
      {'Name': 'city', 'Value': CITY},
      {'Name': 'hostname', 'Value': HOSTNAME}
    ],
    'MeasureName': 'utilization',
    'MeasureValueType': 'MULTI'
  }
  return common_attributes


def prepare_record(current_time):
  record = {
    'Time': str(current_time),
    'MeasureValues': []
  }
  return record


def prepare_measure(measure_name, measure_value):
  measure = {
    'Name': measure_name,
    'Value': str(measure_value),
    'Type': 'DOUBLE'
  }
  return measure
```

```
def write_records(records, common_attributes):
  try:
    result = write_client.write_records(DatabaseName=DATABASE_NAME,
                                        TableName=TABLE_NAME,
                                        CommonAttributes=common_attributes,
                                        Records=records)
    status = result['ResponseMetadata']['HTTPStatusCode']
    print("Processed %d records. WriteRecords HTTPStatusCode: %s" %
        (len(records), status))
  except Exception as err:
    print("Error:", err)


if __name__ == '__main__':

  print("writing data to database {} table {}".format(
    DATABASE_NAME, TABLE_NAME))

  session = boto3.Session()
  write_client = session.client('timestream-write', config=Config(
    read_timeout=20, max_pool_connections=5000, retries={'max_attempts': 10}))
  query_client = session.client('timestream-query') # Not used

  common_attributes = prepare_common_attributes()

  records = []

  while True:

    current_time = int(time.time() * 1000)
    cpu_utilization = psutil.cpu_percent()
    memory_utilization = psutil.virtual_memory().percent
    swap_utilization = psutil.swap_memory().percent
    disk_utilization = psutil.disk_usage('/').percent

    record = prepare_record(current_time)
    record['MeasureValues'].append(prepare_measure('cpu', cpu_utilization))
    record['MeasureValues'].append(prepare_measure('memory', memory_utilization))
    record['MeasureValues'].append(prepare_measure('swap', swap_utilization))
    record['MeasureValues'].append(prepare_measure('disk', disk_utilization))

    records.append(record)
```

```
    print("records {} - cpu {} - memory {} - swap {} - disk {}".format(
      len(records), cpu_utilization, memory_utilization,
      swap_utilization, disk_utilization))

    if len(records) == 100:
      write_records(records, common_attributes)
      records = []

    time.sleep(INTERVAL)
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
  async function writeRecords() {
    console.log("Writing records");
    const currentTime = Date.now().toString(); // Unix time in milliseconds

    const dimensions = [
    {'Name': 'region', 'Value': 'us-east-1'},
    {'Name': 'az', 'Value': 'az1'},
    {'Name': 'hostname', 'Value': 'host1'}
    ];

    const record = {
    'Dimensions': dimensions,
    'MeasureName': 'metrics',
    'MeasureValues': [
      {
        'Name': 'cpu_utilization',
        'Value': '40',
        'Type': 'DOUBLE',
      },
      {
        'Name': 'memory_utilization',
        'Value': '13.5',
        'Type': 'DOUBLE',
      },
      ],
      'MeasureValueType': 'MULTI',
      'Time': currentTime.toString()
    }
```

```
    const records = [record];

    const params = {
    DatabaseName: 'DatabaseName',
    TableName: 'TableName',
    Records: records
    };

    const response = await writeClient.writeRecords(params);

    console.log(response);
  }
```

.NET

```
using System;
using System.IO;
using System.Collections.Generic;
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
using System.Threading.Tasks;

namespace TimestreamDotNetSample
{
  static class MultiMeasureValueConstants
  {
    public const string MultiMeasureValueSampleDb = "multiMeasureValueSampleDb";
    public const string MultiMeasureValueSampleTable =
 "multiMeasureValueSampleTable";
  }

  public class MultiValueAttributesExample
  {
    private readonly AmazonTimestreamWriteClient writeClient;

    public MultiValueAttributesExample(AmazonTimestreamWriteClient writeClient)
    {
      this.writeClient = writeClient;
    }

    public async Task WriteRecordsMultiMeasureValueSingleRecord()
    {
```

```
        Console.WriteLine("Writing records with multi value attributes");

        DateTimeOffset now = DateTimeOffset.UtcNow;
        string currentTimeString = (now.ToUnixTimeMilliseconds()).ToString();

        List<Dimension> dimensions = new List<Dimension>{
          new Dimension { Name = "region", Value = "us-east-1" },
          new Dimension { Name = "az", Value = "az1" },
          new Dimension { Name = "hostname", Value = "host1" }
        };

        var commonAttributes = new Record
        {
          Dimensions = dimensions,
          Time = currentTimeString
        };

        var cpuUtilization = new MeasureValue
        {
          Name = "cpu_utilization",
          Value = "13.6",
          Type = "DOUBLE"
        };

        var memoryUtilization = new MeasureValue
        {
          Name = "memory_utilization",
          Value = "40",
          Type = "DOUBLE"
        };

        var computationalRecord = new Record
        {
          MeasureName = "cpu_memory",
          MeasureValues = new List<MeasureValue> {cpuUtilization, memoryUtilization},
          MeasureValueType = "MULTI"
        };


        List<Record> records = new List<Record>();
        records.Add(computationalRecord);

        try
        {
```

```
        var writeRecordsRequest = new WriteRecordsRequest
        {
          DatabaseName = MultiMeasureValueConstants.MultiMeasureValueSampleDb,
          TableName = MultiMeasureValueConstants.MultiMeasureValueSampleTable,
          Records = records,
          CommonAttributes = commonAttributes
        };
        WriteRecordsResponse response = await
 writeClient.WriteRecordsAsync(writeRecordsRequest);
        Console.WriteLine($"Write records status code:
 {response.HttpStatusCode.ToString()}");
      }
      catch (Exception e)
      {
        Console.WriteLine("Write records failure:" + e.ToString());
      }
    }

    public async Task WriteRecordsMultiMeasureValueMultipleRecords()
    {
      Console.WriteLine("Writing records with multi value attributes mixture type");

      DateTimeOffset now = DateTimeOffset.UtcNow;
      string currentTimeString = (now.ToUnixTimeMilliseconds()).ToString();

      List<Dimension> dimensions = new List<Dimension>{
        new Dimension { Name = "region", Value = "us-east-1" },
        new Dimension { Name = "az", Value = "az1" },
        new Dimension { Name = "hostname", Value = "host1" }
      };

      var commonAttributes = new Record
      {
        Dimensions = dimensions,
        Time = currentTimeString
      };

      var cpuUtilization = new MeasureValue
      {
        Name = "cpu_utilization",
        Value = "13.6",
        Type = "DOUBLE"
      };
```

```
      var memoryUtilization = new MeasureValue
      {
        Name = "memory_utilization",
        Value = "40",
        Type = "DOUBLE"
      };

      var activeCores = new MeasureValue
      {
        Name = "active_cores",
        Value = "4",
        Type = "BIGINT"
      };

      var computationalRecord = new Record
      {
        MeasureName = "computational_utilization",
        MeasureValues = new List<MeasureValue> {cpuUtilization, memoryUtilization,
 activeCores},
        MeasureValueType = "MULTI"
      };

      var aliveRecord = new Record
      {
        MeasureName = "is_healthy",
        MeasureValue = "true",
        MeasureValueType = "BOOLEAN"
      };

      List<Record> records = new List<Record>();
      records.Add(computationalRecord);
      records.Add(aliveRecord);

      try
      {
        var writeRecordsRequest = new WriteRecordsRequest
        {
          DatabaseName = MultiMeasureValueConstants.MultiMeasureValueSampleDb,
          TableName = MultiMeasureValueConstants.MultiMeasureValueSampleTable,
          Records = records,
          CommonAttributes = commonAttributes
        };
        WriteRecordsResponse response = await
 writeClient.WriteRecordsAsync(writeRecordsRequest);
```

```
        Console.WriteLine($"Write records status code:
 {response.HttpStatusCode.ToString()}");
      }
      catch (Exception e)
      {
        Console.WriteLine("Write records failure:" + e.ToString());
      }
    }
  }
}
```

Handling write failures

Writes in Amazon Timestream can fail for one or more of the following reasons:

- There are records with timestamps that lie outside the retention duration of the memory store.
- There are records containing dimensions and/or measures that exceed the Timestream defined limits.
- Amazon Timestream has detected duplicate records. Records are marked as duplicate, when there are multiple records with the same dimensions, timestamps, and measure names but:
  - Measure values are different.
  - Version is not present in the request or the value of version in the new record is equal to or lower than the existing value. If Amazon Timestream rejects data for this reason, the `ExistingVersion` field in the `RejectedRecords` will contain the record's current version as stored in Amazon Timestream. To force an update, you can resend the request with a version for the record set to a value greater than the `ExistingVersion`.

For more information about errors and rejected records, see Errors and RejectedRecord.

If your application receives a `RejectedRecordsException` when attempting to write records to Timestream, you can parse the rejected records to learn more about the write failures as shown below.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
 try {
   WriteRecordsResult writeRecordsResult =
amazonTimestreamWrite.writeRecords(writeRecordsRequest);
   System.out.println("WriteRecords Status: " +
writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
 } catch (RejectedRecordsException e) {
   System.out.println("RejectedRecords: " + e);
   for (RejectedRecord rejectedRecord : e.getRejectedRecords()) {
     System.out.println("Rejected Index " + rejectedRecord.getRecordIndex() + ": "
         + rejectedRecord.getReason());
   }
   System.out.println("Other records were written successfully. ");
 } catch (Exception e) {
   System.out.println("Error: " + e);
 }
```

Java v2

```
   try {
     WriteRecordsResponse writeRecordsResponse =
timestreamWriteClient.writeRecords(writeRecordsRequest);
     System.out.println("writeRecordsWithCommonAttributes Status: " +
writeRecordsResponse.sdkHttpResponse().statusCode());
   } catch (RejectedRecordsException e) {
     System.out.println("RejectedRecords: " + e);
     for (RejectedRecord rejectedRecord : e.rejectedRecords()) {
       System.out.println("Rejected Index " + rejectedRecord.recordIndex() + ": "
           + rejectedRecord.reason());
     }
     System.out.println("Other records were written successfully. ");
   } catch (Exception e) {
     System.out.println("Error: " + e);
   }
```

Go

```
_, err = writeSvc.WriteRecords(writeRecordsInput)

if err != nil {
  fmt.Println("Error:")
  fmt.Println(err)
```

```
} else {
  fmt.Println("Write records is successful")
}
```

Python

```
try:
  result = self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME, Records=records, CommonAttributes=common_attributes)
  print("WriteRecords Status: [%s]" % result['ResponseMetadata']['HTTPStatusCode'])
except self.client.exceptions.RejectedRecordsException as err:
  print("RejectedRecords: ", err)
  for rr in err.response["RejectedRecords"]:
    print("Rejected Index " + str(rr["RecordIndex"]) + ": " + rr["Reason"])
  print("Other records were written successfully. ")
except Exception as err:
  print("Error:", err)
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
await request.promise().then(
    (data) => {
      console.log("Write records successful");
    },
    (err) => {
      console.log("Error writing records:", err);
      if (err.code === 'RejectedRecordsException') {
        const responsePayload =
 JSON.parse(request.response.httpResponse.body.toString());
        console.log("RejectedRecords: ", responsePayload.RejectedRecords);
        console.log("Other records were written successfully. ");
      }
    }
  );
```

.NET

```
  try
  {
```

```
    var writeRecordsRequest = new WriteRecordsRequest
    {
      DatabaseName = Constants.DATABASE_NAME,
      TableName = Constants.TABLE_NAME,
      Records = records,
      CommonAttributes = commonAttributes
    };
    WriteRecordsResponse response = await
 writeClient.WriteRecordsAsync(writeRecordsRequest);
    Console.WriteLine($"Write records status code:
 {response.HttpStatusCode.ToString()}");
  }
  catch (RejectedRecordsException e) {
    Console.WriteLine("RejectedRecordsException:" + e.ToString());
    foreach (RejectedRecord rr in e.RejectedRecords) {
      Console.WriteLine("RecordIndex " + rr.RecordIndex + " : " + rr.Reason);
    }
    Console.WriteLine("Other records were written successfully. ");
  }
  catch (Exception e)
  {
    Console.WriteLine("Write records failure:" + e.ToString());
  }
```

Run query

Topics

- Paginating results
- Parsing result sets
- Accessing the query status

Paginating results

When you run a query, Timestream returns the result set in a paginated manner to optimize the responsiveness of your applications. The code snippet below shows how you can paginate through the result set. You must loop through all the result set pages until you encounter a null value. Pagination tokens expire 3 hours after being issued by Timestream for LiveAnalytics.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    private void runQuery(String queryString) {
        try {
            QueryRequest queryRequest = new QueryRequest();
            queryRequest.setQueryString(queryString);
            QueryResult queryResult = queryClient.query(queryRequest);
            while (true) {
                parseQueryResult(queryResult);
                if (queryResult.getNextToken() == null) {
                    break;
                }
                queryRequest.setNextToken(queryResult.getNextToken());
                queryResult = queryClient.query(queryRequest);
            }
        } catch (Exception e) {
            // Some queries might fail with 500 if the result of a sequence function
has more than 10000 entries
            e.printStackTrace();
        }
    }
```

Java v2

```
    private void runQuery(String queryString) {
        try {
            QueryRequest queryRequest =
QueryRequest.builder().queryString(queryString).build();
            final QueryIterable queryResponseIterator =
timestreamQueryClient.queryPaginator(queryRequest);
            for(QueryResponse queryResponse : queryResponseIterator) {
                parseQueryResult(queryResponse);
            }
        } catch (Exception e) {
            // Some queries might fail with 500 if the result of a sequence function
has more than 10000 entries
```

```
            e.printStackTrace();
        }
    }
```

Go

```
func runQuery(queryPtr *string, querySvc *timestreamquery.TimestreamQuery, f
 *os.File) {
    queryInput := &timestreamquery.QueryInput{
        QueryString: aws.String(*queryPtr),
    }
    fmt.Println("QueryInput:")
    fmt.Println(queryInput)
    // execute the query
    err := querySvc.QueryPages(queryInput,
        func(page *timestreamquery.QueryOutput, lastPage bool) bool {
            // process query response
            queryStatus := page.QueryStatus
            fmt.Println("Current query status:", queryStatus)
            // query response metadata
            // includes column names and types
            metadata := page.ColumnInfo
            // fmt.Println("Metadata:")
            fmt.Println(metadata)
            header := ""
            for i := 0; i < len(metadata); i++ {
                header += *metadata[i].Name
                if i != len(metadata)-1 {
                    header += ", "
                }
            }
            write(f, header)

            // query response data
            fmt.Println("Data:")
            // process rows
            rows := page.Rows
            for i := 0; i < len(rows); i++ {
                data := rows[i].Data
                value := processRowType(data, metadata)
                fmt.Println(value)
                write(f, value)
            }
```

```
            fmt.Println("Number of rows:", len(page.Rows))
            return true
        })
    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    }
}
```

Python

```
    def run_query(self, query_string):
        try:
            page_iterator = self.paginator.paginate(QueryString=query_string)
            for page in page_iterator:
                self._parse_query_result(page)
        except Exception as err:
            print("Exception while running query:", err)
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function getAllRows(query, nextToken) {
    const params = {
        QueryString: query
    };

    if (nextToken) {
        params.NextToken = nextToken;
    }

    await queryClient.query(params).promise()
        .then(
            (response) => {
                parseQueryResult(response);
                if (response.NextToken) {
                    getAllRows(query, response.NextToken);
                }
            },
            (err) => {
                console.error("Error while querying:", err);
```

```
            });
}
```

.NET

```
        private async Task RunQueryAsync(string queryString)
        {
            try
            {
                QueryRequest queryRequest = new QueryRequest();
                queryRequest.QueryString = queryString;
                QueryResponse queryResponse = await
queryClient.QueryAsync(queryRequest);
                while (true)
                {
                    ParseQueryResult(queryResponse);
                    if (queryResponse.NextToken == null)
                    {
                        break;
                    }
                    queryRequest.NextToken = queryResponse.NextToken;
                    queryResponse = await queryClient.QueryAsync(queryRequest);
                }
            } catch(Exception e)
            {
                // Some queries might fail with 500 if the result of a sequence
function has more than 10000 entries
                Console.WriteLine(e.ToString());
            }
        }
```

Parsing result sets

You can use the following code snippets to extract data from the result set. Query results are accessible for up to 24 hours after a query completes.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
   private static final DateTimeFormatter TIMESTAMP_FORMATTER =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.SSSSSSSSS");
   private static final DateTimeFormatter DATE_FORMATTER =
DateTimeFormatter.ofPattern("yyyy-MM-dd");
   private static final DateTimeFormatter TIME_FORMATTER =
DateTimeFormatter.ofPattern("HH:mm:ss.SSSSSSSSS");

   private static final long ONE_GB_IN_BYTES = 1073741824L;

   private void parseQueryResult(QueryResult response) {
       final QueryStatus currentStatusOfQuery = queryResult.getQueryStatus();

       System.out.println("Query progress so far: " +
currentStatusOfQuery.getProgressPercentage() + "%");

       double bytesScannedSoFar = ((double)
currentStatusOfQuery.getCumulativeBytesScanned() / ONE_GB_IN_BYTES);
       System.out.println("Bytes scanned so far: " + bytesScannedSoFar + " GB");

       double bytesMeteredSoFar = ((double)
currentStatusOfQuery.getCumulativeBytesMetered() / ONE_GB_IN_BYTES);
       System.out.println("Bytes metered so far: " + bytesMeteredSoFar + " GB");

       List<ColumnInfo> columnInfo = response.getColumnInfo();
       List<Row> rows = response.getRows();

       System.out.println("Metadata: " + columnInfo);
       System.out.println("Data: ");

       // iterate every row
       for (Row row : rows) {
           System.out.println(parseRow(columnInfo, row));
       }
   }

   private String parseRow(List<ColumnInfo> columnInfo, Row row) {
       List<Datum> data = row.getData();
       List<String> rowOutput = new ArrayList<>();
       // iterate every column per row
       for (int j = 0; j < data.size(); j++) {
           ColumnInfo info = columnInfo.get(j);
           Datum datum = data.get(j);
```

```
            rowOutput.add(parseDatum(info, datum));
        }
        return String.format("{%s}",
rowOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }

    private String parseDatum(ColumnInfo info, Datum datum) {
        if (datum.isNullValue() != null && datum.isNullValue()) {
            return info.getName() + "=" + "NULL";
        }
        Type columnType = info.getType();
        // If the column is of TimeSeries Type
        if (columnType.getTimeSeriesMeasureValueColumnInfo() != null) {
            return parseTimeSeries(info, datum);
        }
        // If the column is of Array Type
        else if (columnType.getArrayColumnInfo() != null) {
            List<Datum> arrayValues = datum.getArrayValue();
            return info.getName() + "=" +
parseArray(info.getType().getArrayColumnInfo(), arrayValues);
        }
        // If the column is of Row Type
        else if (columnType.getRowColumnInfo() != null) {
            List<ColumnInfo> rowColumnInfo = info.getType().getRowColumnInfo();
            Row rowValues = datum.getRowValue();
            return parseRow(rowColumnInfo, rowValues);
        }
        // If the column is of Scalar Type
        else {
            return parseScalarType(info, datum);
        }
    }

    private String parseTimeSeries(ColumnInfo info, Datum datum) {
        List<String> timeSeriesOutput = new ArrayList<>();
        for (TimeSeriesDataPoint dataPoint : datum.getTimeSeriesValue()) {
            timeSeriesOutput.add("{time=" + dataPoint.getTime() + ", value=" +
                    parseDatum(info.getType().getTimeSeriesMeasureValueColumnInfo(),
dataPoint.getValue()) + "}");
        }
        return String.format("[%s]",
timeSeriesOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }
```

```
    private String parseScalarType(ColumnInfo info, Datum datum) {
        switch (ScalarType.fromValue(info.getType().getScalarType())) {
            case VARCHAR:
                return parseColumnName(info) + datum.getScalarValue();
            case BIGINT:
                Long longValue = Long.valueOf(datum.getScalarValue());
                return parseColumnName(info) + longValue;
            case INTEGER:
                Integer intValue = Integer.valueOf(datum.getScalarValue());
                return parseColumnName(info) + intValue;
            case BOOLEAN:
                Boolean booleanValue = Boolean.valueOf(datum.getScalarValue());
                return parseColumnName(info) + booleanValue;
            case DOUBLE:
                Double doubleValue = Double.valueOf(datum.getScalarValue());
                return parseColumnName(info) + doubleValue;
            case TIMESTAMP:
                return parseColumnName(info) +
 LocalDateTime.parse(datum.getScalarValue(), TIMESTAMP_FORMATTER);
            case DATE:
                return parseColumnName(info) +
 LocalDate.parse(datum.getScalarValue(), DATE_FORMATTER);
            case TIME:
                return parseColumnName(info) +
 LocalTime.parse(datum.getScalarValue(), TIME_FORMATTER);
            case INTERVAL_DAY_TO_SECOND:
            case INTERVAL_YEAR_TO_MONTH:
                return parseColumnName(info) + datum.getScalarValue();
            case UNKNOWN:
                return parseColumnName(info) + datum.getScalarValue();
            default:
                throw new IllegalArgumentException("Given type is not valid: " +
 info.getType().getScalarType());
        }
    }

    private String parseColumnName(ColumnInfo info) {
        return info.getName() == null ? "" : info.getName() + "=";
    }

    private String parseArray(ColumnInfo arrayColumnInfo, List<Datum> arrayValues) {
        List<String> arrayOutput = new ArrayList<>();
        for (Datum datum : arrayValues) {
            arrayOutput.add(parseDatum(arrayColumnInfo, datum));
```

```
        }
        return String.format("[%s]",
arrayOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }
```

Java v2

```
    private static final long ONE_GB_IN_BYTES = 1073741824L;

    private void parseQueryResult(QueryResponse response) {
        final QueryStatus currentStatusOfQuery = response.queryStatus();

        System.out.println("Query progress so far: " +
currentStatusOfQuery.progressPercentage() + "%");

        double bytesScannedSoFar = ((double)
currentStatusOfQuery.cumulativeBytesScanned() / ONE_GB_IN_BYTES);
        System.out.println("Bytes scanned so far: " + bytesScannedSoFar + " GB");

        double bytesMeteredSoFar = ((double)
currentStatusOfQuery.cumulativeBytesMetered() / ONE_GB_IN_BYTES);
        System.out.println("Bytes metered so far: " + bytesMeteredSoFar + " GB");

        List<ColumnInfo> columnInfo = response.columnInfo();
        List<Row> rows = response.rows();

        System.out.println("Metadata: " + columnInfo);
        System.out.println("Data: ");

        // iterate every row
        for (Row row : rows) {
            System.out.println(parseRow(columnInfo, row));
        }
    }

    private String parseRow(List<ColumnInfo> columnInfo, Row row) {
        List<Datum> data = row.data();
        List<String> rowOutput = new ArrayList<>();
        // iterate every column per row
        for (int j = 0; j < data.size(); j++) {
            ColumnInfo info = columnInfo.get(j);
            Datum datum = data.get(j);
            rowOutput.add(parseDatum(info, datum));
```

```
        }
        return String.format("{%s}",
rowOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }

    private String parseDatum(ColumnInfo info, Datum datum) {
        if (datum.nullValue() != null && datum.nullValue()) {
            return info.name() + "=" + "NULL";
        }
        Type columnType = info.type();
        // If the column is of TimeSeries Type
        if (columnType.timeSeriesMeasureValueColumnInfo() != null) {
            return parseTimeSeries(info, datum);
        }
        // If the column is of Array Type
        else if (columnType.arrayColumnInfo() != null) {
            List<Datum> arrayValues = datum.arrayValue();
            return info.name() + "=" + parseArray(info.type().arrayColumnInfo(),
arrayValues);
        }
        // If the column is of Row Type
        else if (columnType.rowColumnInfo() != null &&
columnType.rowColumnInfo().size() > 0) {
            List<ColumnInfo> rowColumnInfo = info.type().rowColumnInfo();
            Row rowValues = datum.rowValue();
            return parseRow(rowColumnInfo, rowValues);
        }
        // If the column is of Scalar Type
        else {
            return parseScalarType(info, datum);
        }
    }

    private String parseTimeSeries(ColumnInfo info, Datum datum) {
        List<String> timeSeriesOutput = new ArrayList<>();
        for (TimeSeriesDataPoint dataPoint : datum.timeSeriesValue()) {
            timeSeriesOutput.add("{time=" + dataPoint.time() + ", value=" +
                    parseDatum(info.type().timeSeriesMeasureValueColumnInfo(),
dataPoint.value()) + "}");
        }
        return String.format("[%s]",
timeSeriesOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }
```

```
    private String parseScalarType(ColumnInfo info, Datum datum) {
        return parseColumnName(info) + datum.scalarValue();
    }

    private String parseColumnName(ColumnInfo info) {
        return info.name() == null ? "" : info.name() + "=";
    }

    private String parseArray(ColumnInfo arrayColumnInfo, List<Datum> arrayValues) {
        List<String> arrayOutput = new ArrayList<>();
        for (Datum datum : arrayValues) {
            arrayOutput.add(parseDatum(arrayColumnInfo, datum));
        }
        return String.format("[%s]",
 arrayOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }
```

Go

```
func processScalarType(data *timestreamquery.Datum) string {
    return *data.ScalarValue
}

func processTimeSeriesType(data []*timestreamquery.TimeSeriesDataPoint, columnInfo
 *timestreamquery.ColumnInfo) string {
    value := ""
    for k := 0; k < len(data); k++ {
        time := data[k].Time
        value += *time + ":"
        if columnInfo.Type.ScalarType != nil {
            value += processScalarType(data[k].Value)
        } else if columnInfo.Type.ArrayColumnInfo != nil {
            value += processArrayType(data[k].Value.ArrayValue,
 columnInfo.Type.ArrayColumnInfo)
        } else if columnInfo.Type.RowColumnInfo != nil {
            value += processRowType(data[k].Value.RowValue.Data,
 columnInfo.Type.RowColumnInfo)
        } else {
            fail("Bad data type")
        }
        if k != len(data)-1 {
            value += ", "
        }
```

```
    }
    return value
}

func processArrayType(datumList []*timestreamquery.Datum, columnInfo
 *timestreamquery.ColumnInfo) string {
    value := ""
    for k := 0; k < len(datumList); k++ {
        if columnInfo.Type.ScalarType != nil {
            value += processScalarType(datumList[k])
        } else if columnInfo.Type.TimeSeriesMeasureValueColumnInfo != nil {
            value += processTimeSeriesType(datumList[k].TimeSeriesValue,
 columnInfo.Type.TimeSeriesMeasureValueColumnInfo)
        } else if columnInfo.Type.ArrayColumnInfo != nil {
            value += "["
            value += processArrayType(datumList[k].ArrayValue,
 columnInfo.Type.ArrayColumnInfo)
            value += "]"
        } else if columnInfo.Type.RowColumnInfo != nil {
            value += "["
            value += processRowType(datumList[k].RowValue.Data,
 columnInfo.Type.RowColumnInfo)
            value += "]"
        } else {
            fail("Bad column type")
        }

        if k != len(datumList)-1 {
            value += ", "
        }
    }
    return value
}

func processRowType(data []*timestreamquery.Datum, metadata
 []*timestreamquery.ColumnInfo) string {
    value := ""
    for j := 0; j < len(data); j++ {
        if metadata[j].Type.ScalarType != nil {
            // process simple data types
            value += processScalarType(data[j])
        } else if metadata[j].Type.TimeSeriesMeasureValueColumnInfo != nil {
            // fmt.Println("Timeseries measure value column info")
            // fmt.Println(metadata[j].Type.TimeSeriesMeasureValueColumnInfo.Type)
```

```
            datapointList := data[j].TimeSeriesValue
            value += "["
            value += processTimeSeriesType(datapointList,
 metadata[j].Type.TimeSeriesMeasureValueColumnInfo)
            value += "]"
        } else if metadata[j].Type.ArrayColumnInfo != nil {
            columnInfo := metadata[j].Type.ArrayColumnInfo
            // fmt.Println("Array column info")
            // fmt.Println(columnInfo)
            datumList := data[j].ArrayValue
            value += "["
            value += processArrayType(datumList, columnInfo)
            value += "]"
        } else if metadata[j].Type.RowColumnInfo != nil {
            columnInfo := metadata[j].Type.RowColumnInfo
            datumList := data[j].RowValue.Data
            value += "["
            value += processRowType(datumList, columnInfo)
            value += "]"
        } else {
            panic("Bad column type")
        }
        // comma seperated column values
        if j != len(data)-1 {
            value += ", "
        }
    }
    return value
}
```

Python

```
    def _parse_query_result(self, query_result):
        query_status = query_result["QueryStatus"]

        progress_percentage = query_status["ProgressPercentage"]
        print(f"Query progress so far: {progress_percentage}%")

        bytes_scanned = float(query_status["CumulativeBytesScanned"]) /
 ONE_GB_IN_BYTES
        print(f"Data scanned so far: {bytes_scanned} GB")
```

```
        bytes_metered = float(query_status["CumulativeBytesMetered"]) /
 ONE_GB_IN_BYTES
        print(f"Data metered so far: {bytes_metered} GB")

        column_info = query_result['ColumnInfo']

        print("Metadata: %s" % column_info)
        print("Data: ")
        for row in query_result['Rows']:
            print(self._parse_row(column_info, row))

    def _parse_row(self, column_info, row):
        data = row['Data']
        row_output = []
        for j in range(len(data)):
            info = column_info[j]
            datum = data[j]
            row_output.append(self._parse_datum(info, datum))

        return "{%s}" % str(row_output)

    def _parse_datum(self, info, datum):
        if datum.get('NullValue', False):
            return "%s=NULL" % info['Name'],

        column_type = info['Type']

        # If the column is of TimeSeries Type
        if 'TimeSeriesMeasureValueColumnInfo' in column_type:
            return self._parse_time_series(info, datum)

        # If the column is of Array Type
        elif 'ArrayColumnInfo' in column_type:
            array_values = datum['ArrayValue']
            return "%s=%s" % (info['Name'], self._parse_array(info['Type']
['ArrayColumnInfo'], array_values))

        # If the column is of Row Type
        elif 'RowColumnInfo' in column_type:
            row_column_info = info['Type']['RowColumnInfo']
            row_values = datum['RowValue']
            return self._parse_row(row_column_info, row_values)

        # If the column is of Scalar Type
```

```
        else:
            return self._parse_column_name(info) + datum['ScalarValue']

    def _parse_time_series(self, info, datum):
        time_series_output = []
        for data_point in datum['TimeSeriesValue']:
            time_series_output.append("{time=%s, value=%s}"
                                      % (data_point['Time'],
                                         self._parse_datum(info['Type']
['TimeSeriesMeasureValueColumnInfo'],
                                                           data_point['Value'])))
        return "[%s]" % str(time_series_output)

    def _parse_array(self, array_column_info, array_values):
        array_output = []
        for datum in array_values:
            array_output.append(self._parse_datum(array_column_info, datum))

        return "[%s]" % str(array_output)

    @staticmethod
    def _parse_column_name(info):
        if 'Name' in info:
            return info['Name'] + "="
        else:
            return ""
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
function parseQueryResult(response) {
    const queryStatus = response.QueryStatus;
    console.log("Current query status: " + JSON.stringify(queryStatus));

    const columnInfo = response.ColumnInfo;
    const rows = response.Rows;

    console.log("Metadata: " + JSON.stringify(columnInfo));
    console.log("Data: ");

    rows.forEach(function (row) {
```

```
        console.log(parseRow(columnInfo, row));
    });
}

function parseRow(columnInfo, row) {
    const data = row.Data;
    const rowOutput = [];

    var i;
    for ( i = 0; i < data.length; i++ ) {
        info = columnInfo[i];
        datum = data[i];
        rowOutput.push(parseDatum(info, datum));
    }

    return `{${rowOutput.join(", ")}}`
}

function parseDatum(info, datum) {
    if (datum.NullValue != null && datum.NullValue === true) {
        return `${info.Name}=NULL`;
    }

    const columnType = info.Type;

    // If the column is of TimeSeries Type
    if (columnType.TimeSeriesMeasureValueColumnInfo != null) {
        return parseTimeSeries(info, datum);
    }
    // If the column is of Array Type
    else if (columnType.ArrayColumnInfo != null) {
        const arrayValues = datum.ArrayValue;
        return `${info.Name}=${parseArray(info.Type.ArrayColumnInfo, arrayValues)}`;
    }
    // If the column is of Row Type
    else if (columnType.RowColumnInfo != null) {
        const rowColumnInfo = info.Type.RowColumnInfo;
        const rowValues = datum.RowValue;
        return parseRow(rowColumnInfo, rowValues);
    }
    // If the column is of Scalar Type
    else {
        return parseScalarType(info, datum);
    }
```

```
}

function parseTimeSeries(info, datum) {
    const timeSeriesOutput = [];
    datum.TimeSeriesValue.forEach(function (dataPoint) {
        timeSeriesOutput.push(`{time=${dataPoint.Time}, value=
${parseDatum(info.Type.TimeSeriesMeasureValueColumnInfo, dataPoint.Value)}}`)
    });

    return `[${timeSeriesOutput.join(", ")}]`
}

function parseScalarType(info, datum) {
    return parseColumnName(info) + datum.ScalarValue;
}

function parseColumnName(info) {
    return info.Name == null ? "" : `${info.Name}=`;
}

function parseArray(arrayColumnInfo, arrayValues) {
    const arrayOutput = [];
    arrayValues.forEach(function (datum) {
        arrayOutput.push(parseDatum(arrayColumnInfo, datum));
    });
    return `[${arrayOutput.join(", ")}]`
}
```

.NET

```
        private void ParseQueryResult(QueryResponse response)
        {
            List<ColumnInfo> columnInfo = response.ColumnInfo;
            var options = new JsonSerializerOptions
            {
                IgnoreNullValues = true
            };
            List<String> columnInfoStrings = columnInfo.ConvertAll(x =>
 JsonSerializer.Serialize(x, options));
            List<Row> rows = response.Rows;

            QueryStatus queryStatus = response.QueryStatus;
```

```
            Console.WriteLine("Current Query status:" +
JsonSerializer.Serialize(queryStatus, options));

            Console.WriteLine("Metadata:" + string.Join(",", columnInfoStrings));
            Console.WriteLine("Data:");

            foreach (Row row in rows)
            {
                Console.WriteLine(ParseRow(columnInfo, row));
            }
        }

        private string ParseRow(List<ColumnInfo> columnInfo, Row row)
        {
            List<Datum> data = row.Data;
            List<string> rowOutput = new List<string>();
            for (int j = 0; j < data.Count; j++)
            {
                ColumnInfo info = columnInfo[j];
                Datum datum = data[j];
                rowOutput.Add(ParseDatum(info, datum));
            }
            return $"{{{string.Join(",", rowOutput)}}}";
        }

        private string ParseDatum(ColumnInfo info, Datum datum)
        {
            if (datum.NullValue)
            {
                return $"{info.Name}=NULL";
            }

            Amazon.TimestreamQuery.Model.Type columnType = info.Type;
            if (columnType.TimeSeriesMeasureValueColumnInfo != null)
            {
                return ParseTimeSeries(info, datum);
            }
            else if (columnType.ArrayColumnInfo != null)
            {
                List<Datum> arrayValues = datum.ArrayValue;
                return $"{info.Name}={ParseArray(info.Type.ArrayColumnInfo,
arrayValues)}";
            }
```

```
            else if (columnType.RowColumnInfo != null &&
columnType.RowColumnInfo.Count > 0)
            {
                List<ColumnInfo> rowColumnInfo = info.Type.RowColumnInfo;
                Row rowValue = datum.RowValue;
                return ParseRow(rowColumnInfo, rowValue);
            }
            else
            {
                return ParseScalarType(info, datum);
            }
        }

        private string ParseTimeSeries(ColumnInfo info, Datum datum)
        {
            var timeseriesString = datum.TimeSeriesValue
                .Select(value => $"{{time={value.Time},
value={ParseDatum(info.Type.TimeSeriesMeasureValueColumnInfo, value.Value)}}}")
                .Aggregate((current, next) => current + "," + next);

            return $"[{timeseriesString}]";
        }

        private string ParseScalarType(ColumnInfo info, Datum datum)
        {
            return ParseColumnName(info) + datum.ScalarValue;
        }

        private string ParseColumnName(ColumnInfo info)
        {
            return info.Name == null ? "" : (info.Name + "=");
        }

        private string ParseArray(ColumnInfo arrayColumnInfo, List<Datum>
arrayValues)
        {
            return $"[{arrayValues.Select(value => ParseDatum(arrayColumnInfo,
value)).Aggregate((current, next) => current + "," + next)}]";
        }
```

Accessing the query status

You can access the query status through `QueryResponse`, which contains information about progress of a query, the bytes scanned by a query and the bytes metered by a query. The `bytesMetered` and `bytesScanned` values are cumulative and continuously updated while paging query results. You can use this information to understand the bytes scanned by an individual query and also use it to make certain decisions. For example, assuming that the query price is $0.01 per GB scanned, you may want to cancel queries that exceed $25 per query, or X GB. The code snippet below shows how this can be done.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
   private static final long ONE_GB_IN_BYTES = 1073741824L;
   private static final double QUERY_COST_PER_GB_IN_DOLLARS = 0.01; // Assuming the
price of query is $0.01 per GB

   public void cancelQueryBasedOnQueryStatus() {
       System.out.println("Starting query: " + SELECT_ALL_QUERY);
       QueryRequest queryRequest = new QueryRequest();
       queryRequest.setQueryString(SELECT_ALL_QUERY);
       QueryResult queryResult = queryClient.query(queryRequest);

       while (true) {
           final QueryStatus currentStatusOfQuery = queryResult.getQueryStatus();
           System.out.println("Query progress so far: " +
currentStatusOfQuery.getProgressPercentage() + "%");
           double bytesMeteredSoFar = ((double)
currentStatusOfQuery.getCumulativeBytesMetered() / ONE_GB_IN_BYTES);
           System.out.println("Bytes metered so far: " + bytesMeteredSoFar + "
GB");
           // Cancel query if its costing more than 1 cent
           if (bytesMeteredSoFar * QUERY_COST_PER_GB_IN_DOLLARS > 0.01) {
               cancelQuery(queryResult);
               break;
           }
```

```
            if (queryResult.getNextToken() == null) {
                break;
            }
            queryRequest.setNextToken(queryResult.getNextToken());
            queryResult = queryClient.query(queryRequest);
        }
    }
```

Java v2

```
    private static final long ONE_GB_IN_BYTES = 1073741824L;
    private static final double QUERY_COST_PER_GB_IN_DOLLARS = 0.01; // Assuming the
 price of query is $0.01 per GB

    public void cancelQueryBasedOnQueryStatus() {
        System.out.println("Starting query: " + SELECT_ALL_QUERY);
        QueryRequest queryRequest =
 QueryRequest.builder().queryString(SELECT_ALL_QUERY).build();

        final QueryIterable queryResponseIterator =
 timestreamQueryClient.queryPaginator(queryRequest);
        for(QueryResponse queryResponse : queryResponseIterator) {
            final QueryStatus currentStatusOfQuery = queryResponse.queryStatus();
            System.out.println("Query progress so far: " +
 currentStatusOfQuery.progressPercentage() + "%");
            double bytesMeteredSoFar = ((double)
 currentStatusOfQuery.cumulativeBytesMetered() / ONE_GB_IN_BYTES);
            System.out.println("Bytes metered so far: " + bytesMeteredSoFar + "GB");
            // Cancel query if its costing more than 1 cent
            if (bytesMeteredSoFar * QUERY_COST_PER_GB_IN_DOLLARS > 0.01) {
                cancelQuery(queryResponse);
                break;
            }
        }
    }
```

Go

```
const OneGbInBytes = 1073741824
// Assuming the price of query is $0.01 per GB
const QueryCostPerGbInDollars = 0.01
```

```
func cancelQueryBasedOnQueryStatus(queryPtr *string, querySvc
 *timestreamquery.TimestreamQuery, f *os.File) {
    queryInput := &timestreamquery.QueryInput{
        QueryString: aws.String(*queryPtr),
    }
    fmt.Println("QueryInput:")
    fmt.Println(queryInput)
    // execute the query
    err := querySvc.QueryPages(queryInput,
        func(page *timestreamquery.QueryOutput, lastPage bool) bool {
            // process query response
            queryStatus := page.QueryStatus
            fmt.Println("Current query status:", queryStatus)
            bytes_metered := float64(*queryStatus.CumulativeBytesMetered) /
 float64(ONE_GB_IN_BYTES)
            if bytes_metered * QUERY_COST_PER_GB_IN_DOLLARS > 0.01 {
                cancelQuery(page, querySvc)
                return true
            }
            // query response metadata
            // includes column names and types
            metadata := page.ColumnInfo
            // fmt.Println("Metadata:")
            fmt.Println(metadata)
            header := ""
            for i := 0; i < len(metadata); i++ {
                header += *metadata[i].Name
                if i != len(metadata)-1 {
                    header += ", "
                }
            }
            write(f, header)

            // query response data
            fmt.Println("Data:")
            // process rows
            rows := page.Rows
            for i := 0; i < len(rows); i++ {
                data := rows[i].Data
                value := processRowType(data, metadata)
                fmt.Println(value)
                write(f, value)
            }
            fmt.Println("Number of rows:", len(page.Rows))
```

```
            return true
        })
    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    }
}
```

Python

```
ONE_GB_IN_BYTES = 1073741824
# Assuming the price of query is $0.01 per GB
QUERY_COST_PER_GB_IN_DOLLARS = 0.01

    def cancel_query_based_on_query_status(self):
        try:
            print("Starting query: " + self.SELECT_ALL)
            page_iterator = self.paginator.paginate(QueryString=self.SELECT_ALL)
            for page in page_iterator:
                query_status = page["QueryStatus"]
                progress_percentage = query_status["ProgressPercentage"]
                print("Query progress so far: " + str(progress_percentage) + "%")
                bytes_metered = query_status["CumulativeBytesMetered"] /
 self.ONE_GB_IN_BYTES
                print("Bytes Metered so far: " + str(bytes_metered) + " GB")
                if bytes_metered * self.QUERY_COST_PER_GB_IN_DOLLARS > 0.01:
                    self.cancel_query_for(page)
                    break
        except Exception as err:
            print("Exception while running query:", err)
            traceback.print_exc(file=sys.stderr)
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
function parseQueryResult(response) {
    const queryStatus = response.QueryStatus;
    console.log("Current query status: " + JSON.stringify(queryStatus));

    const columnInfo = response.ColumnInfo;
    const rows = response.Rows;
```

```
    console.log("Metadata: " + JSON.stringify(columnInfo));
    console.log("Data: ");

    rows.forEach(function (row) {
        console.log(parseRow(columnInfo, row));
    });
}

function parseRow(columnInfo, row) {
    const data = row.Data;
    const rowOutput = [];

    var i;
    for ( i = 0; i < data.length; i++ ) {
        info = columnInfo[i];
        datum = data[i];
        rowOutput.push(parseDatum(info, datum));
    }

    return `{${rowOutput.join(", ")}}`
}

function parseDatum(info, datum) {
    if (datum.NullValue != null && datum.NullValue === true) {
        return `${info.Name}=NULL`;
    }

    const columnType = info.Type;

    // If the column is of TimeSeries Type
    if (columnType.TimeSeriesMeasureValueColumnInfo != null) {
        return parseTimeSeries(info, datum);
    }
    // If the column is of Array Type
    else if (columnType.ArrayColumnInfo != null) {
        const arrayValues = datum.ArrayValue;
        return `${info.Name}=${parseArray(info.Type.ArrayColumnInfo, arrayValues)}`;
    }
    // If the column is of Row Type
    else if (columnType.RowColumnInfo != null) {
        const rowColumnInfo = info.Type.RowColumnInfo;
        const rowValues = datum.RowValue;
        return parseRow(rowColumnInfo, rowValues);
```

```
    }
    // If the column is of Scalar Type
    else {
        return parseScalarType(info, datum);
    }
}

function parseTimeSeries(info, datum) {
    const timeSeriesOutput = [];
    datum.TimeSeriesValue.forEach(function (dataPoint) {
        timeSeriesOutput.push(`{time=${dataPoint.Time}, value=
${parseDatum(info.Type.TimeSeriesMeasureValueColumnInfo, dataPoint.Value)}}`)
    });

    return `[${timeSeriesOutput.join(", ")}]`
}

function parseScalarType(info, datum) {
    return parseColumnName(info) + datum.ScalarValue;
}

function parseColumnName(info) {
    return info.Name == null ? "" : `${info.Name}=`;
}

function parseArray(arrayColumnInfo, arrayValues) {
    const arrayOutput = [];
    arrayValues.forEach(function (datum) {
        arrayOutput.push(parseDatum(arrayColumnInfo, datum));
    });
    return `[${arrayOutput.join(", ")}]`
}
```

.NET

```
private static readonly long ONE_GB_IN_BYTES = 1073741824L;
private static readonly double QUERY_COST_PER_GB_IN_DOLLARS = 0.01; // Assuming the
 price of query is $0.01 per GB

private async Task CancelQueryBasedOnQueryStatus(string queryString)
{
    try
    {
```

```
        QueryRequest queryRequest = new QueryRequest();
        queryRequest.QueryString = queryString;
        QueryResponse queryResponse = await queryClient.QueryAsync(queryRequest);
        while (true)
        {
            QueryStatus queryStatus = queryResponse.QueryStatus;
            double bytesMeteredSoFar = ((double)
 queryStatus.CumulativeBytesMetered / ONE_GB_IN_BYTES);
            // Cancel query if its costing more than 1 cent
            if (bytesMeteredSoFar * QUERY_COST_PER_GB_IN_DOLLARS > 0.01)
            {
                await CancelQuery(queryResponse);
                break;
            }

            ParseQueryResult(queryResponse);
            if (queryResponse.NextToken == null)
            {
                break;
            }
            queryRequest.NextToken = queryResponse.NextToken;
            queryResponse = await queryClient.QueryAsync(queryRequest);
       }
    } catch(Exception e)
    {
        // Some queries might fail with 500 if the result of a sequence function has
 more than 10000 entries
        Console.WriteLine(e.ToString());
    }
}
```

For additional details on how to cancel a query, see Cancel query.

Run UNLOAD query

The following code examples call an UNLOAD query. For information about UNLOAD, see Using UNLOAD to export query results to S3 from Timestream for LiveAnalytics. For examples of UNLOAD queries, see Example use case for UNLOAD from Timestream for LiveAnalytics.

Topics

- Build and run an UNLOAD query

- Parse UNLOAD response, and get row count, manifest link, and metadata link
- Read and parse manifest content
- Read and parse metadata content

Build and run an UNLOAD query

Java

```
// When you have a SELECT like below

String QUERY_1 = "SELECT user_id, ip_address, event, session_id, measure_name, time,
 query, quantity, product_id, channel FROM "
        + DATABASE_NAME + "." + UNLOAD_TABLE_NAME
        + " WHERE time BETWEEN ago(2d) AND now()";

// You can construct UNLOAD query as follows
UnloadQuery unloadQuery = UnloadQuery.builder()
        .selectQuery(QUERY_1)
        .bucketName("timestream-sample-<region>-<accountId>")
        .resultsPrefix("without_partition")
        .format(CSV)
        .compression(UnloadQuery.Compression.GZIP)
        .build();
QueryResult unloadResult = runQuery(unloadQuery.getUnloadQuery());

// Run UNLOAD query (Similar to how you run SELECT query)
// https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.pagination
    private QueryResult runQuery(String queryString) {
        QueryResult queryResult = null;
        try {
            QueryRequest queryRequest = new QueryRequest();
            queryRequest.setQueryString(queryString);
            queryResult = queryClient.query(queryRequest);
            while (true) {
                parseQueryResult(queryResult);
                if (queryResult.getNextToken() == null) {
                    break;
                }
                queryRequest.setNextToken(queryResult.getNextToken());
                queryResult = queryClient.query(queryRequest);
            }
```

```
        } catch (Exception e) {
            // Some queries might fail with 500 if the result of a sequence function
 has more than 10000 entries
            e.printStackTrace();
        }
        return queryResult;
    }

// Utility that helps to construct UNLOAD query

@Builder
static class UnloadQuery {
    private String selectQuery;
    private String bucketName;
    private String resultsPrefix;
    private Format format;
    private Compression compression;
    private EncryptionType encryptionType;
    private List<String> partitionColumns;
    private String kmsKey;
    private Character csvFieldDelimiter;
    private Character csvEscapeCharacter;

    public String getUnloadQuery() {
        String destination = constructDestination();
        String withClause = constructOptionalParameters();
        return String.format("UNLOAD (%s) TO '%s' %s", selectQuery, destination,
 withClause);
    }

    private String constructDestination() {
        return "s3://" + this.bucketName + "/" + this.resultsPrefix + "/";
    }

    private String constructOptionalParameters() {
        boolean isOptionalParametersPresent = Objects.nonNull(format)
                || Objects.nonNull(compression)
                || Objects.nonNull(encryptionType)
                || Objects.nonNull(partitionColumns)
                || Objects.nonNull(kmsKey)
                || Objects.nonNull(csvFieldDelimiter)
                || Objects.nonNull(csvEscapeCharacter);

        String withClause = "";
```

```
        if (isOptionalParametersPresent) {
            StringJoiner optionalParameters = new StringJoiner(",");
            if (Objects.nonNull(format)) {
                optionalParameters.add("format = '" + format + "'");
            }
            if (Objects.nonNull(compression)) {
                optionalParameters.add("compression = '" + compression + "'");
            }
            if (Objects.nonNull(encryptionType)) {
                optionalParameters.add("encryption = '" + encryptionType + "'");
            }
            if (Objects.nonNull(kmsKey)) {
                optionalParameters.add("kms_key = '" + kmsKey + "'");
            }
            if (Objects.nonNull(csvFieldDelimiter)) {
                optionalParameters.add("field_delimiter = '" + csvFieldDelimiter +
"'");
            }
            if (Objects.nonNull(csvEscapeCharacter)) {
                optionalParameters.add("escaped_by = '" + csvEscapeCharacter + "'");
            }
            if (Objects.nonNull(partitionColumns) && !partitionColumns.isEmpty()) {
                final StringJoiner partitionedByList = new StringJoiner(",");
                partitionColumns.forEach(column -> partitionedByList.add("'" +
column + "'"));
                optionalParameters.add(String.format("partitioned_by = ARRAY[%s]",
partitionedByList));
            }
            withClause = String.format("WITH (%s)", optionalParameters);
        }
        return withClause;
    }

    public enum Format {
        CSV, PARQUET
    }

    public enum Compression {
        GZIP, NONE
    }

    public enum EncryptionType {
        SSE_S3, SSE_KMS
    }
```

```
    @Override
    public String toString() {
        return getUnloadQuery();
    }
}
```

Java v2

```
// When you have a SELECT like below

String QUERY_1 = "SELECT user_id, ip_address, event, session_id, measure_name, time,
 query, quantity, product_id, channel FROM "
        + DATABASE_NAME + "." + UNLOAD_TABLE_NAME
        + " WHERE time BETWEEN ago(2d) AND now()";

//You can construct UNLOAD query as follows
UnloadQuery unloadQuery = UnloadQuery.builder()
        .selectQuery(QUERY_1)
        .bucketName("timestream-sample-<region>-<accountId>")
        .resultsPrefix("without_partition")
        .format(CSV)
        .compression(UnloadQuery.Compression.GZIP)
        .build();

QueryResponse unloadResponse = runQuery(unloadQuery.getUnloadQuery());


// Run UNLOAD query (Similar to how you run SELECT query)
// https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.pagination
private QueryResponse runQuery(String queryString) {
   QueryResponse finalResponse = null;
    try {
        QueryRequest queryRequest =
 QueryRequest.builder().queryString(queryString).build();
        final QueryIterable queryResponseIterator =
 timestreamQueryClient.queryPaginator(queryRequest);
        for(QueryResponse queryResponse : queryResponseIterator) {
            parseQueryResult(queryResponse);
           finalResponse = queryResponse;
        }
    } catch (Exception e) {
```

```
        // Some queries might fail with 500 if the result of a sequence function has
 more than 10000 entries
        e.printStackTrace();
    }
   return finalResponse;
}

// Utility that helps to construct UNLOAD query
@Builder
static class UnloadQuery {
    private String selectQuery;
    private String bucketName;
    private String resultsPrefix;
    private Format format;
    private Compression compression;
    private EncryptionType encryptionType;
    private List<String> partitionColumns;
    private String kmsKey;
    private Character csvFieldDelimiter;
    private Character csvEscapeCharacter;

    public String getUnloadQuery() {
        String destination = constructDestination();
        String withClause = constructOptionalParameters();
        return String.format("UNLOAD (%s) TO '%s' %s", selectQuery, destination,
 withClause);
    }

    private String constructDestination() {
        return "s3://" + this.bucketName + "/" + this.resultsPrefix + "/";
    }

    private String constructOptionalParameters() {
        boolean isOptionalParametersPresent = Objects.nonNull(format)
                || Objects.nonNull(compression)
                || Objects.nonNull(encryptionType)
                || Objects.nonNull(partitionColumns)
                || Objects.nonNull(kmsKey)
                || Objects.nonNull(csvFieldDelimiter)
                || Objects.nonNull(csvEscapeCharacter);

        String withClause = "";
        if (isOptionalParametersPresent) {
            StringJoiner optionalParameters = new StringJoiner(",");
```

```
            if (Objects.nonNull(format)) {
                optionalParameters.add("format = '" + format + "'");
            }
            if (Objects.nonNull(compression)) {
                optionalParameters.add("compression = '" + compression + "'");
            }
            if (Objects.nonNull(encryptionType)) {
                optionalParameters.add("encryption = '" + encryptionType + "'");
            }
            if (Objects.nonNull(kmsKey)) {
                optionalParameters.add("kms_key = '" + kmsKey + "'");
            }
            if (Objects.nonNull(csvFieldDelimiter)) {
                optionalParameters.add("field_delimiter = '" + csvFieldDelimiter +
"'");
            }
            if (Objects.nonNull(csvEscapeCharacter)) {
                optionalParameters.add("escaped_by = '" + csvEscapeCharacter + "'");
            }
            if (Objects.nonNull(partitionColumns) && !partitionColumns.isEmpty()) {
                final StringJoiner partitionedByList = new StringJoiner(",");
                partitionColumns.forEach(column -> partitionedByList.add("'" +
column + "'"));
                optionalParameters.add(String.format("partitioned_by = ARRAY[%s]",
partitionedByList));
            }
            withClause = String.format("WITH (%s)", optionalParameters);
        }
        return withClause;
    }

    public enum Format {
        CSV, PARQUET
    }

    public enum Compression {
        GZIP, NONE
    }

    public enum EncryptionType {
        SSE_S3, SSE_KMS
    }

    @Override
```

```
    public String toString() {
        return getUnloadQuery();
    }
}
```

Go

```
// When you have a SELECT like below
var Query = "SELECT user_id, ip_address, event, session_id, measure_name, time,
 query, quantity, product_id, channel FROM "
+ *databaseName + "." + *tableName + " WHERE time BETWEEN ago(2d) AND now()"

// You can construct UNLOAD query as follows
var unloadQuery = UnloadQuery{
    Query: "SELECT user_id, ip_address, session_id, measure_name, time, query,
 quantity, product_id, channel, event FROM " + *databaseName + "." + *tableName +
    " WHERE time BETWEEN ago(2d) AND now()",
    Partitioned_by: []string{},
    Compression: "GZIP",
    Format: "CSV",
    S3Location: bucketName,
    ResultPrefix: "without_partition",
}


// Run UNLOAD query (Similar to how you run SELECT query)
// https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.pagination

queryInput := &timestreamquery.QueryInput{
    QueryString: build_query(unloadQuery),
}

err := querySvc.QueryPages(queryInput,
    func(page *timestreamquery.QueryOutput, lastPage bool) bool {
        if (lastPage) {
            var response = parseQueryResult(page)
            var unloadFiles = getManifestAndMetadataFiles(s3Svc, response)
            displayColumns(unloadFiles, unloadQuery.Partitioned_by)
            displayResults(s3Svc, unloadFiles)
        }
        return true
    })
```

```
if err != nil {
    fmt.Println("Error:")
    fmt.Println(err)
}

// Utility that helps to construct UNLOAD query
type UnloadQuery struct {
    Query string
    Partitioned_by []string
    Format string
    S3Location string
    ResultPrefix string
    Compression string
}

func build_query(unload_query UnloadQuery) *string {
    var query_results_s3_path = "'s3://" + unload_query.S3Location + "/" +
 unload_query.ResultPrefix + "/'"
    var query = "UNLOAD(" + unload_query.Query + ") TO " + query_results_s3_path + "
 WITH ( "
    if (len(unload_query.Partitioned_by) > 0) {
        query = query + "partitioned_by=ARRAY["
        for i, column := range unload_query.Partitioned_by {
            if i == 0 {
                query = query + "'" + column + "'"
            } else {
                query = query + ",'" + column + "'"
            }
        }
        query = query + "],"
    }
    query = query + " format='" + unload_query.Format + "', "
    query = query + "  compression='" + unload_query.Compression + "')"
    fmt.Println(query)
    return aws.String(query)
}
```

Python

```
# When you have a SELECT like below
QUERY_1 = "SELECT user_id, ip_address, event, session_id, measure_name, time, query,
 quantity, product_id, channel FROM "
```

```
        + database_name + "." + table_name + " WHERE time BETWEEN ago(2d) AND now()"
# You can construct UNLOAD query as follows
UNLOAD_QUERY_1 = UnloadQuery(QUERY_1, "timestream-sample-<region>-<accountId>",
 "without_partition", "CSV", "GZIP", "")

# Run UNLOAD query (Similar to how you run SELECT query)
# https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.pagination
def run_query(self, query_string):
    try:
        page_iterator = self.paginator.paginate(QueryString=UNLOAD_QUERY_1)
    except Exception as err:
        print("Exception while running query:", err)

# Utility that helps to construct UNLOAD query
class UnloadQuery:
    def __init__(self, query, s3_bucket_location, results_prefix, format,
 compression , partition_by):
        self.query = query
        self.s3_bucket_location = s3_bucket_location
        self.results_prefix = results_prefix
        self.format = format
        self.compression = compression
        self.partition_by = partition_by

    def build_query(self):
        query_results_s3_path = "'s3://" + self.s3_bucket_location + "/" +
 self.results_prefix + "/'"
        unload_query = "UNLOAD("
        unload_query = unload_query + self.query
        unload_query = unload_query + ") "
        unload_query = unload_query + " TO " + query_results_s3_path
        unload_query = unload_query + " WITH ( "

        if(len(self.partition_by) > 0) :
            unload_query = unload_query + " partitioned_by = ARRAY" +
 str(self.partition_by) + ","

        unload_query = unload_query + " format='" + self.format  + "', "
        unload_query = unload_query + "  compression='" + self.compression + "')"

        return unload_query
```

Node.js

```
// When you have a SELECT like below
QUERY_1 = "SELECT user_id, ip_address, event, session_id, measure_name, time, query,
 quantity, product_id, channel FROM "
        + database_name + "." + table_name + " WHERE time BETWEEN ago(2d) AND now()"
// You can construct UNLOAD query as follows
UNLOAD_QUERY_1 = new UnloadQuery(QUERY_1, "timestream-sample-<region>-<accountId>",
 "without_partition", "CSV", "GZIP", "")


// Run UNLOAD query (Similar to how you run SELECT query)
// https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.pagination

async runQuery(query = UNLOAD_QUERY_1, nextToken) {
    const params = new QueryCommand({
        QueryString: query
    });

    if (nextToken) {
        params.NextToken = nextToken;
    }

    await queryClient.send(params).then(
            (response) => {
                if (response.NextToken) {
                    runQuery(queryClient, query, response.NextToken);
                } else {
                    await parseAndDisplayResults(response);
                }
            },
            (err) => {
                console.error("Error while querying:", err);
            });
}


class UnloadQuery {
    constructor(query, s3_bucket_location, results_prefix, format, compression ,
 partition_by) {
        this.query = query;
        this.s3_bucket_location = s3_bucket_location
        this.results_prefix = results_prefix
```

```
        this.format = format
        this.compression = compression
        this.partition_by = partition_by
    }

    buildQuery() {
        const query_results_s3_path = "'s3://" + this.s3_bucket_location + "/" +
 this.results_prefix + "/'"
        let unload_query = "UNLOAD("
        unload_query = unload_query + this.query
        unload_query = unload_query + ") "
        unload_query = unload_query + " TO " + query_results_s3_path
        unload_query = unload_query + " WITH ( "

        if(this.partition_by.length > 0) {
            let partitionBy = ""
            this.partition_by.forEach((str, i) => {
                partitionBy = partitionBy + (i ? ",'" : "'") + str + "'"
            })
            unload_query = unload_query + " partitioned_by = ARRAY[" + partitionBy +
 "],"
        }
        unload_query = unload_query + " format='" + this.format  + "', "
        unload_query = unload_query + "  compression='" + this.compression + "')"

        return unload_query
    }
}
```

Parse UNLOAD response, and get row count, manifest link, and metadata link

Java

```
// Parsing UNLOAD query response is similar to how you parse SELECT query response:
// https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.parsing

// But unlike SELECT, UNLOAD only has 1 row * 3 columns outputed
// (rows, metadataFile, manifestFile) => (BIGINT, VARCHAR, VARCHAR)

public UnloadResponse parseResult(QueryResult queryResult) {
    Map<String, String> outputMap = new HashMap<>();
```

```
    for (int i = 0; i < queryResult.getColumnInfo().size(); i++) {
        outputMap.put(queryResult.getColumnInfo().get(i).getName(),
                queryResult.getRows().get(0).getData().get(i).getScalarValue());

    }
    return new UnloadResponse(outputMap);
}

@Getter
class UnloadResponse {
    private final String metadataFile;
    private final String manifestFile;
    private final int rows;

    public UnloadResponse(Map<String, String> unloadResponse) {
        this.metadataFile = unloadResponse.get("metadataFile");
        this.manifestFile = unloadResponse.get("manifestFile");
        this.rows = Integer.parseInt(unloadResponse.get("rows"));
    }
}
```

Java v2

```
// Parsing UNLOAD query response is similar to how you parse SELECT query response:
// https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.parsing

// But unlike SELECT, UNLOAD only has 1 row * 3 columns outputed
// (rows, metadataFile, manifestFile) => (BIGINT, VARCHAR, VARCHAR)

public UnloadResponse parseResult(QueryResponse queryResponse) {
    Map<String, String> outputMap = new HashMap<>();
    for (int i = 0; i < queryResponse.columnInfo().size(); i++) {
        outputMap.put(queryResponse.columnInfo().get(i).name(),
                queryResponse.rows().get(0).data().get(i).scalarValue());

    }
    return new UnloadResponse(outputMap);
}

@Getter
class UnloadResponse {
    private final String metadataFile;
```

```
    private final String manifestFile;
    private final int rows;

    public UnloadResponse(Map<String, String> unloadResponse) {
        this.metadataFile = unloadResponse.get("metadataFile");
        this.manifestFile = unloadResponse.get("manifestFile");
        this.rows = Integer.parseInt(unloadResponse.get("rows"));
    }
}
```

Go

```
// Parsing UNLOAD query response is similar to how you parse SELECT query response:
// https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.parsing

// But unlike SELECT, UNLOAD only has 1 row * 3 columns outputed
// (rows, metadataFile, manifestFile) => (BIGINT, VARCHAR, VARCHAR)

func parseQueryResult(queryOutput *timestreamquery.QueryOutput) map[string]string {
    var columnInfo = queryOutput.ColumnInfo;
    fmt.Println("ColumnInfo", columnInfo)
    fmt.Println("QueryId", queryOutput.QueryId)
    fmt.Println("QueryStatus", queryOutput.QueryStatus)
    return parseResponse(columnInfo, queryOutput.Rows[0])
}

func parseResponse(columnInfo []*timestreamquery.ColumnInfo, row
 *timestreamquery.Row) map[string]string {
    var datum = row.Data
    response := make(map[string]string)
    for i, column := range columnInfo {
        response[*column.Name] = *datum[i].ScalarValue
    }
    return response
}
```

Python

```
# Parsing UNLOAD query response is similar to how you parse SELECT query response:
# https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.parsing
```

```
# But unlike SELECT, UNLOAD only has 1 row * 3 columns outputed
# (rows, metadataFile, manifestFile) => (BIGINT, VARCHAR, VARCHAR)

for page in page_iterator:
  last_page = page
response = self._parse_unload_query_result(last_page)

def _parse_unload_query_result(self, query_result):
    column_info = query_result['ColumnInfo']

    print("ColumnInfo: %s" % column_info)
    print("QueryId: %s" % query_result['QueryId'])
    print("QueryStatus:%s" % query_result['QueryStatus'])
    return self.parse_unload_response(column_info, query_result['Rows'][0])

def parse_unload_response(self, column_info, row):
    response = {}
    data = row['Data']
    for i, column in enumerate(column_info):
        response[column['Name']] = data[i]['ScalarValue']
   print("Rows: %s" % response['rows'])
   print("Metadata File location: %s" % response['metadataFile'])
   print("Manifest File location: %s" % response['manifestFile'])
   return response
```

Node.js

```
# Parsing UNLOAD query response is similar to how you parse SELECT query response:
# https://docs.aws.amazon.com/timestream/latest/developerguide/code-samples.run-
query.html#code-samples.run-query.parsing

# But unlike SELECT, UNLOAD only has 1 row * 3 columns outputed
# (rows, metadataFile, manifestFile) => (BIGINT, VARCHAR, VARCHAR)

async parseAndDisplayResults(data, query) {
    const columnInfo = data['ColumnInfo'];
    console.log("ColumnInfo:", columnInfo)
    console.log("QueryId: %s", data['QueryId'])
    console.log("QueryStatus:", data['QueryStatus'])
    await this.parseResponse(columnInfo, data['Rows'][0], query)
}

async parseResponse(columnInfo, row, query) {
```

```
    let response = {}
    const data = row['Data']
    columnInfo.forEach((column, i) => {
        response[column['Name']] = data[i]['ScalarValue']
    })

   console.log("Manifest file", response['manifestFile']);
   console.log("Metadata file", response['metadataFile']);

   return response
}
```

Read and parse manifest content

Java

```
// Read and parse manifest content
public UnloadManifest getUnloadManifest(UnloadResponse unloadResponse) throws
 IOException {
   AmazonS3URI s3URI = new AmazonS3URI(unloadResponse.getManifestFile());
    S3Object s3Object = s3Client.getObject(s3URI.getBucket(), s3URI.getKey());
    String manifestFileContent = new
 String(IOUtils.toByteArray(s3Object.getObjectContent()), StandardCharsets.UTF_8);
    return new Gson().fromJson(manifestFileContent, UnloadManifest.class);
}

class UnloadManifest {
    @Getter
    public class FileMetadata {
        long content_length_in_bytes;
        long row_count;
    }

    @Getter
    public class ResultFile {
        String url;
        FileMetadata file_metadata;
    }

    @Getter
    public class QueryMetadata {
        long total_content_length_in_bytes;
```

```
        long total_row_count;
        String result_format;
        String result_version;
    }

    @Getter
    public class Author {
        String name;
        String manifest_file_version;
    }

    @Getter
    private List<ResultFile> result_files;
    @Getter
    private QueryMetadata query_metadata;
    @Getter
    private Author author;
}
```

Java v2

```
// Read and parse manifest content
public UnloadManifest getUnloadManifest(UnloadResponse unloadResponse) throws
 URISyntaxException {
    // Space needs to encoded to use S3 parseUri function
    S3Uri s3Uri =
 s3Utilities.parseUri(URI.create(unloadResponse.getManifestFile().replace(" ",
 "%20")));
    ResponseBytes<GetObjectResponse> objectBytes =
 s3Client.getObjectAsBytes(GetObjectRequest.builder()
            .bucket(s3Uri.bucket().orElseThrow(() -> new
 URISyntaxException(unloadResponse.getManifestFile(), "Invalid S3 URI")))
            .key(s3Uri.key().orElseThrow(() -> new
 URISyntaxException(unloadResponse.getManifestFile(), "Invalid S3 URI")))
            .build());
    String manifestFileContent = new String(objectBytes.asByteArray(),
 StandardCharsets.UTF_8);
    return new Gson().fromJson(manifestFileContent, UnloadManifest.class);
}

class UnloadManifest {
    @Getter
    public class FileMetadata {
```

```
        long content_length_in_bytes;
        long row_count;
    }

    @Getter
    public class ResultFile {
        String url;
        FileMetadata file_metadata;
    }

    @Getter
    public class QueryMetadata {
        long total_content_length_in_bytes;
        long total_row_count;
        String result_format;
        String result_version;
    }

    @Getter
    public class Author {
        String name;
        String manifest_file_version;
    }

    @Getter
    private List<ResultFile> result_files;
    @Getter
    private QueryMetadata query_metadata;
    @Getter
    private Author author;
}
```

Go

```
// Read and parse manifest content

func getManifestFile(s3Svc *s3.S3, response map[string]string) Manifest {
    var manifestBuf = getObject(s3Svc, response["manifestFile"])
    var manifest Manifest
    json.Unmarshal(manifestBuf.Bytes(), &manifest)
    return manifest
}
```

```
func getObject(s3Svc *s3.S3, s3Uri string) *bytes.Buffer {
    u,_ := url.Parse(s3Uri)
    getObjectInput := &s3.GetObjectInput{
        Key:    aws.String(u.Path),
        Bucket: aws.String(u.Host),
    }
    getObjectOutput, err := s3Svc.GetObject(getObjectInput)
    if err != nil {
        fmt.Println("Error: %s\n", err.Error())
    }
    buf := new(bytes.Buffer)
    buf.ReadFrom(getObjectOutput.Body)
    return buf
}

// Unload's Manifest structure

type Manifest struct {
    Author interface{}
    Query_metadata map[string]any
    Result_files  []struct {
        File_metadata interface{}
        Url string
    }
}}
```

Python

```
def __get_manifest_file(self, response):
    manifest = self.get_object(response['manifestFile']).read().decode('utf-8')
    parsed_manifest = json.loads(manifest)
    print("Manifest contents: \n%s" % parsed_manifest)

def get_object(self, uri):
    try:
        bucket, key = uri.replace("s3://", "").split("/", 1)
        s3_client = boto3.client('s3', region_name=<region>)
        response = s3_client.get_object(Bucket=bucket, Key=key)
        return response['Body']
    except Exception as err:
        print("Failed to get the object for URI:", uri)
        raise err
```

Node.js

```
// Read and parse manifest content

async getManifestFile(response) {
    let manifest;
    await this.getS3Object(response['manifestFile']).then(
        (data) => {
            manifest = JSON.parse(data);
        }
    );
    return manifest;
}

async getS3Object(uri) {
    const {bucketName, key} = this.getBucketAndKey(uri);
    const params = new GetObjectCommand({
        Bucket: bucketName,
        Key: key
    })
    const response = await this.s3Client.send(params);
    return await response.Body.transformToString();
}

getBucketAndKey(uri) {
    const [bucketName] = uri.replace("s3://", "").split("/", 1);
    const key = uri.replace("s3://", "").split('/').slice(1).join('/');
    return {bucketName, key};
}
```

Read and parse metadata content

Java

```
// Read and parse metadata content
public UnloadMetadata getUnloadMetadata(UnloadResponse unloadResponse) throws
 IOException {
   AmazonS3URI s3URI = new AmazonS3URI(unloadResponse.getMetadataFile());
   S3Object s3Object = s3Client.getObject(s3URI.getBucket(), s3URI.getKey());
    String metadataFileContent = new
 String(IOUtils.toByteArray(s3Object.getObjectContent()), StandardCharsets.UTF_8);
    final Gson gson = new GsonBuilder()
```

```
            .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE)
            .create();
    return gson.fromJson(metadataFileContent, UnloadMetadata.class);
}

class UnloadMetadata {
    @JsonProperty("ColumnInfo")
    List<ColumnInfo> columnInfo;
    @JsonProperty("Author")
    Author author;

    @Data
    public class Author {
        @JsonProperty("Name")
        String name;
        @JsonProperty("MetadataFileVersion")
        String metadataFileVersion;
    }
}
```

Java v2

```
// Read and parse metadata content

public UnloadMetadata getUnloadMetadata(UnloadResponse unloadResponse) throws
 URISyntaxException {
   // Space needs to encoded to use S3 parseUri function
    S3Uri s3Uri =
 s3Utilities.parseUri(URI.create(unloadResponse.getMetadataFile().replace(" ",
 "%20")));
    ResponseBytes<GetObjectResponse> objectBytes =
 s3Client.getObjectAsBytes(GetObjectRequest.builder()
            .bucket(s3Uri.bucket().orElseThrow(() -> new
 URISyntaxException(unloadResponse.getMetadataFile(), "Invalid S3 URI")))
            .key(s3Uri.key().orElseThrow(() -> new
 URISyntaxException(unloadResponse.getMetadataFile(), "Invalid S3 URI")))
            .build());
    String metadataFileContent = new String(objectBytes.asByteArray(),
 StandardCharsets.UTF_8);
    final Gson gson = new GsonBuilder()
            .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE)
            .create();
    return gson.fromJson(metadataFileContent, UnloadMetadata.class);
```

```
}

class UnloadMetadata {
    @JsonProperty("ColumnInfo")
    List<ColumnInfo> columnInfo;
    @JsonProperty("Author")
    Author author;

    @Data
    public class Author {
        @JsonProperty("Name")
        String name;
        @JsonProperty("MetadataFileVersion")
        String metadataFileVersion;
    }
}
```

Go

```
// Read and parse metadata content

func getMetadataFile(s3Svc *s3.S3, response map[string]string) Metadata {
    var metadataBuf = getObject(s3Svc, response["metadataFile"])
    var metadata Metadata
    json.Unmarshal(metadataBuf.Bytes(), &metadata)
    return metadata
}

func getObject(s3Svc *s3.S3, s3Uri string) *bytes.Buffer {
    u,_ := url.Parse(s3Uri)
    getObjectInput := &s3.GetObjectInput{
        Key:    aws.String(u.Path),
        Bucket: aws.String(u.Host),
    }
    getObjectOutput, err := s3Svc.GetObject(getObjectInput)
    if err != nil {
        fmt.Println("Error: %s\n", err.Error())
    }
    buf := new(bytes.Buffer)
    buf.ReadFrom(getObjectOutput.Body)
    return buf
}
```

```
// Unload's Metadata structure

type Metadata struct {
    Author interface{}
    ColumnInfo []struct {
        Name string
        Type map[string]string
    }
}
```

Python

```
def __get_metadata_file(self, response):
    metadata = self.get_object(response['metadataFile']).read().decode('utf-8')
    parsed_metadata = json.loads(metadata)
   print("Metadata contents: \n%s" % parsed_metadata)

def get_object(self, uri):
    try:
        bucket, key = uri.replace("s3://", "").split("/", 1)
        s3_client = boto3.client('s3', region_name=<region>)
        response = s3_client.get_object(Bucket=bucket, Key=key)
        return response['Body']
    except Exception as err:
        print("Failed to get the object for URI:", uri)
        raise err
```

Node.js

```
// Read and parse metadata content
async getMetadataFile(response) {
    let metadata;
    await this.getS3Object(response['metadataFile']).then(
        (data) => {
            metadata = JSON.parse(data);
        }
    );
    return metadata;
}

async getS3Object(uri) {
```

```
    const {bucketName, key} = this.getBucketAndKey(uri);
    const params = new GetObjectCommand({
        Bucket: bucketName,
        Key: key
    })
    const response = await this.s3Client.send(params);
    return await response.Body.transformToString();
}

getBucketAndKey(uri) {
    const [bucketName] = uri.replace("s3://", "").split("/", 1);
    const key = uri.replace("s3://", "").split('/').slice(1).join('/');
    return {bucketName, key};
}
```

Cancel query

You can use the following code snippets to cancel a query.

> **Note**
>
> These code snippets are based on full sample applications on GitHub. For more information about how to get started with the sample applications, see Sample application.

Java

```
    public void cancelQuery() {
        System.out.println("Starting query: " + SELECT_ALL_QUERY);
        QueryRequest queryRequest = new QueryRequest();
        queryRequest.setQueryString(SELECT_ALL_QUERY);
        QueryResult queryResult = queryClient.query(queryRequest);

        System.out.println("Cancelling the query: " + SELECT_ALL_QUERY);
        final CancelQueryRequest cancelQueryRequest = new CancelQueryRequest();
        cancelQueryRequest.setQueryId(queryResult.getQueryId());
        try {
            queryClient.cancelQuery(cancelQueryRequest);
            System.out.println("Query has been successfully cancelled");
        } catch (Exception e) {
```

```
            System.out.println("Could not cancel the query: " + SELECT_ALL_QUERY + "
 = " + e);
        }
    }
```

Java v2

```
    public void cancelQuery() {
        System.out.println("Starting query: " + SELECT_ALL_QUERY);
        QueryRequest queryRequest =
 QueryRequest.builder().queryString(SELECT_ALL_QUERY).build();
        QueryResponse queryResponse = timestreamQueryClient.query(queryRequest);

        System.out.println("Cancelling the query: " + SELECT_ALL_QUERY);
        final CancelQueryRequest cancelQueryRequest = CancelQueryRequest.builder()
                .queryId(queryResponse.queryId()).build();
        try {
            timestreamQueryClient.cancelQuery(cancelQueryRequest);
            System.out.println("Query has been successfully cancelled");
        } catch (Exception e) {
            System.out.println("Could not cancel the query: " + SELECT_ALL_QUERY + "
 = " + e);
        }
    }
```

Go

```
cancelQueryInput := &timestreamquery.CancelQueryInput{
      QueryId: aws.String(*queryOutput.QueryId),
  }

  fmt.Println("Submitting cancellation for the query")
  fmt.Println(cancelQueryInput)

  // submit the query
  cancelQueryOutput, err := querySvc.CancelQuery(cancelQueryInput)

  if err != nil {
      fmt.Println("Error:")
      fmt.Println(err)
  } else {
    fmt.Println("Query has been cancelled successfully")
    fmt.Println(cancelQueryOutput)
```

```
  }
```

Python

```
    def cancel_query(self):
        print("Starting query: " + self.SELECT_ALL)
        result = self.client.query(QueryString=self.SELECT_ALL)
        print("Cancelling query: " + self.SELECT_ALL)
        try:
            self.client.cancel_query(QueryId=result['QueryId'])
            print("Query has been successfully cancelled")
        except Exception as err:
            print("Cancelling query failed:", err)
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function tryCancelQuery() {
    const params = {
        QueryString: SELECT_ALL_QUERY
    };
    console.log(`Running query: ${SELECT_ALL_QUERY}`);

    await queryClient.query(params).promise()
        .then(
            async (response) => {
                await cancelQuery(response.QueryId);
            },
            (err) => {
                console.error("Error while executing select all query:", err);
            });
}

async function cancelQuery(queryId) {
    const cancelParams = {
        QueryId: queryId
    };
    console.log(`Sending cancellation for query: ${SELECT_ALL_QUERY}`);
    await queryClient.cancelQuery(cancelParams).promise()
        .then(
            (response) => {
```

```
                console.log("Query has been cancelled successfully");
            },
            (err) => {
                console.error("Error while cancelling select all:", err);
            });
}
```

.NET

```
        public async Task CancelQuery()
        {
            Console.WriteLine("Starting query: " + SELECT_ALL_QUERY);
            QueryRequest queryRequest = new QueryRequest();
            queryRequest.QueryString = SELECT_ALL_QUERY;
            QueryResponse queryResponse = await
 queryClient.QueryAsync(queryRequest);

            Console.WriteLine("Cancelling query: " + SELECT_ALL_QUERY);
            CancelQueryRequest cancelQueryRequest = new CancelQueryRequest();
            cancelQueryRequest.QueryId = queryResponse.QueryId;

            try
            {
                await queryClient.CancelQueryAsync(cancelQueryRequest);
                Console.WriteLine("Query has been successfully cancelled.");
            } catch(Exception e)
            {
                Console.WriteLine("Could not cancel the query: " + SELECT_ALL_QUERY
 + " = " + e);
            }
        }
```

Create batch load task

You can use the following code snippets to create batch load tasks.

Java

```
package com.example.tryit;

import java.util.Arrays;
```

```
import
 software.amazon.awssdk.services.timestreamwrite.model.CreateBatchLoadTaskRequest;
import
 software.amazon.awssdk.services.timestreamwrite.model.CreateBatchLoadTaskResponse;
import software.amazon.awssdk.services.timestreamwrite.model.DataModel;
import software.amazon.awssdk.services.timestreamwrite.model.DataModelConfiguration;
import
 software.amazon.awssdk.services.timestreamwrite.model.DataSourceConfiguration;
import
 software.amazon.awssdk.services.timestreamwrite.model.DataSourceS3Configuration;
import software.amazon.awssdk.services.timestreamwrite.model.DimensionMapping;
import
 software.amazon.awssdk.services.timestreamwrite.model.MultiMeasureAttributeMapping;
import software.amazon.awssdk.services.timestreamwrite.model.MultiMeasureMappings;
import software.amazon.awssdk.services.timestreamwrite.model.ReportConfiguration;
import software.amazon.awssdk.services.timestreamwrite.model.ReportS3Configuration;
import software.amazon.awssdk.services.timestreamwrite.model.ScalarMeasureValueType;
import software.amazon.awssdk.services.timestreamwrite.model.TimeUnit;
import software.amazon.awssdk.services.timestreamwrite.TimestreamWriteClient;

public class BatchLoadExample {
    public static final String DATABASE_NAME = <database name>;
    public static final String TABLE_NAME = <table name>;
    public static final String INPUT_BUCKET = <S3 location>;
    public static final String INPUT_OBJECT_KEY_PREFIX = <CSV filename>;
    public static final String REPORT_BUCKET = <S3 location>;
    public static final long HT_TTL_HOURS = 24L;
    public static final long CT_TTL_DAYS = 7L;

    TimestreamWriteClient amazonTimestreamWrite;

    public BatchLoadExample(TimestreamWriteClient client) {
        this.amazonTimestreamWrite = client;
    }

    public String createBatchLoadTask() {
        System.out.println("Creating batch load task");

        CreateBatchLoadTaskRequest request = CreateBatchLoadTaskRequest.builder()
                .dataModelConfiguration(DataModelConfiguration.builder()
                        .dataModel(DataModel.builder()
                                .timeColumn("timestamp")
                                .timeUnit(TimeUnit.SECONDS)
                                .dimensionMappings(Arrays.asList(
```

```
                                        DimensionMapping.builder()
                                                .sourceColumn("vehicle")
                                                .build(),
                                        DimensionMapping.builder()
                                                .sourceColumn("registration")
                                                .destinationColumn("license")
                                                .build()))
                                .multiMeasureMappings(MultiMeasureMappings.builder()
                                        .targetMultiMeasureName("mva_measure_name")

.multiMeasureAttributeMappings(Arrays.asList(

MultiMeasureAttributeMapping.builder()
                                                        .sourceColumn("wgt")

.targetMultiMeasureAttributeName("weight")

.measureValueType(ScalarMeasureValueType.DOUBLE)
                                                        .build(),

MultiMeasureAttributeMapping.builder()
                                                        .sourceColumn("spd")

.targetMultiMeasureAttributeName("speed")

.measureValueType(ScalarMeasureValueType.DOUBLE)
                                                        .build(),

MultiMeasureAttributeMapping.builder()
                                                        .sourceColumn("fuel")

.measureValueType(ScalarMeasureValueType.DOUBLE)
                                                        .build(),

MultiMeasureAttributeMapping.builder()
                                                        .sourceColumn("miles")

.measureValueType(ScalarMeasureValueType.DOUBLE)
                                                        .build()))
                                        .build())
                                .build())
                        .build())
                .dataSourceConfiguration(DataSourceConfiguration.builder()
                        .dataSourceS3Configuration(
```

```
                                DataSourceS3Configuration.builder()
                                        .bucketName(INPUT_BUCKET)
                                        .objectKeyPrefix(INPUT_OBJECT_KEY_PREFIX)
                                        .build())
                        .dataFormat("CSV")
                        .build())
                .reportConfiguration(ReportConfiguration.builder()
                        .reportS3Configuration(ReportS3Configuration.builder()
                                .bucketName(REPORT_BUCKET)
                                .build())
                        .build())
                .targetDatabaseName(DATABASE_NAME)
                .targetTableName(TABLE_NAME)
                .build();
        try {
            final CreateBatchLoadTaskResponse createBatchLoadTaskResponse =
 amazonTimestreamWrite.createBatchLoadTask(request);
            String taskId = createBatchLoadTaskResponse.taskId();
            System.out.println("Successfully created batch load task: " + taskId);
            return taskId;
        } catch (Exception e) {
            System.out.println("Failed to create batch load task: " + e);
            throw e;
        }
    }
}
```

Go

```
package main

import (
 "fmt"
 "context"
 "log"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/timestreamwrite"
 "github.com/aws/aws-sdk-go-v2/service/timestreamwrite/types"
)

func main() {
```

```
 customResolver := aws.EndpointResolverWithOptionsFunc(func(service, region string,
 options ...interface{})(aws.Endpoint, error) {
  if service == timestreamwrite.ServiceID &&  region == "us-west-2" {
      return aws.Endpoint{
          PartitionID:   "aws",
          URL:           <URL>,
          SigningRegion: "us-west-2",
      }, nil
  }
  return aws.Endpoint{}, &  aws.EndpointNotFoundError{}
 })

 cfg, err := config.LoadDefaultConfig(context.TODO(),
 config.WithEndpointResolverWithOptions(customResolver), config.WithRegion("us-
west-2"))

 if err != nil {
    log.Fatalf("failed to load configuration, %v", err)
 }

 client := timestreamwrite.NewFromConfig(cfg)

 response, err := client.CreateBatchLoadTask(context.TODO(), &
 timestreamwrite.CreateBatchLoadTaskInput{
           TargetDatabaseName: aws.String("BatchLoadExampleDatabase"),
           TargetTableName: aws.String("BatchLoadExampleTable"),
           RecordVersion: aws.Int64(1),
           DataModelConfiguration: &  types.DataModelConfiguration{
               DataModel: &  types.DataModel{
                   TimeColumn: aws.String("timestamp"),
                   TimeUnit: types.TimeUnitMilliseconds,
                   DimensionMappings: []types.DimensionMapping{
                       {
                           SourceColumn: aws.String("registration"),
                           DestinationColumn: aws.String("license"),
                       },
                   },
                   MultiMeasureMappings: &  types.MultiMeasureMappings{
                       TargetMultiMeasureName: aws.String("mva_measure_name"),
                       MultiMeasureAttributeMappings:
 []types.MultiMeasureAttributeMapping{
                           {
                               SourceColumn: aws.String("wgt"),
```

```
                                TargetMultiMeasureAttributeName:
 aws.String("weight"),
                                MeasureValueType:
 types.ScalarMeasureValueTypeDouble,
                            },
                            {
                                SourceColumn: aws.String("spd"),
                                TargetMultiMeasureAttributeName:
 aws.String("speed"),
                                MeasureValueType:
 types.ScalarMeasureValueTypeDouble,
                            },
                            {
                                SourceColumn: aws.String("fuel_consumption"),
                                TargetMultiMeasureAttributeName: aws.String("fuel"),
                                MeasureValueType:
 types.ScalarMeasureValueTypeDouble,
                            },
                        },
                    },
                },
            },
           DataSourceConfiguration: &  types.DataSourceConfiguration{
                DataSourceS3Configuration: &  types.DataSourceS3Configuration{
                    BucketName: aws.String("test-batch-load-west-2"),
                    ObjectKeyPrefix: aws.String("sample.csv"),
                },
                DataFormat: types.BatchLoadDataFormatCsv,
            },
           ReportConfiguration: & types.ReportConfiguration{
                ReportS3Configuration: & types.ReportS3Configuration{
                    BucketName: aws.String("test-batch-load-report-west-2"),
                    EncryptionOption: types.S3EncryptionOptionSseS3,
                },
            },
 })

 fmt.Println(aws.ToString(response.TaskId))
}
```

Python

```
import boto3
```

```
from botocore.config import Config

INGEST_ENDPOINT = "<URL>"
REGION = "us-west-2"
HT_TTL_HOURS = 24
CT_TTL_DAYS = 7
DATABASE_NAME = "<database name>"
TABLE_NAME = "<table name>"
INPUT_BUCKET_NAME = "<S3 location>"
INPUT_OBJECT_KEY_PREFIX = "<CSV file name>"
REPORT_BUCKET_NAME = "<S3 location>"


def create_batch_load_task(client, database_name, table_name, input_bucket_name,
 input_object_key_prefix, report_bucket_name):
    try:
        result = client.create_batch_load_task(TargetDatabaseName=database_name,
 TargetTableName=table_name,
                                                DataModelConfiguration={"DataModel":
 {
                                                    "TimeColumn": "timestamp",
                                                    "TimeUnit": "SECONDS",
                                                    "DimensionMappings": [
                                                        {
                                                            "SourceColumn": "vehicle"
                                                        },
                                                        {
                                                            "SourceColumn":
 "registration",
                                                            "DestinationColumn":
 "license"
                                                        }
                                                    ],
                                                    "MultiMeasureMappings": {
                                                        "TargetMultiMeasureName":
 "metrics",

 "MultiMeasureAttributeMappings": [
                                                            {
                                                                "SourceColumn":
 "wgt",
                                                                "MeasureValueType":
 "DOUBLE"
                                                            },
```

```
                                                                    {
                                                                        "SourceColumn":
"spd",
                                                                        "MeasureValueType":
"DOUBLE"
                                                                    },
                                                                    {
                                                                        "SourceColumn":
"fuel_consumption",

"TargetMultiMeasureAttributeName": "fuel",
                                                                        "MeasureValueType":
"DOUBLE"
                                                                    },
                                                                    {
                                                                        "SourceColumn":
"miles",
                                                                        "MeasureValueType":
"DOUBLE"
                                                                    }
                                                            ]}
                                                    }
                                                    },
                                                    DataSourceConfiguration={
                                                        "DataSourceS3Configuration": {
                                                            "BucketName":
input_bucket_name,
                                                            "ObjectKeyPrefix":
input_object_key_prefix
                                                        },
                                                        "DataFormat": "CSV"
                                                    },
                                                    ReportConfiguration={
                                                        "ReportS3Configuration": {
                                                            "BucketName":
report_bucket_name,
                                                            "EncryptionOption": "SSE_S3"
                                                        }
                                                    }
                                                    )

       task_id = result["TaskId"]
       print("Successfully created batch load task: ", task_id)
       return task_id
```

```
    except Exception as err:
        print("Create batch load task job failed:", err)
        return None


if __name__ == '__main__':
    session = boto3.Session()

    write_client = session.client('timestream-write',
                                  endpoint_url=INGEST_ENDPOINT, region_name=REGION,
                                  config=Config(read_timeout=20,
 max_pool_connections=5000, retries={'max_attempts': 10}))

    task_id = create_batch_load_task(write_client, DATABASE_NAME, TABLE_NAME,
                                     INPUT_BUCKET_NAME, INPUT_OBJECT_KEY_PREFIX,
 REPORT_BUCKET_NAME)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

For API details, see Class CreateBatchLoadCommand and CreateBatchLoadTask.

```
import { TimestreamWriteClient, CreateBatchLoadTaskCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "us-west-2", endpoint:
 "https://gamma-ingest-cell3.timestream.us-west-2.amazonaws.com" });

const params = {
    TargetDatabaseName: "BatchLoadExampleDatabase",
    TargetTableName: "BatchLoadExampleTable",
 RecordVersion: 1,
 DataModelConfiguration: {
  DataModel: {
   TimeColumn: "timestamp",
            TimeUnit: "MILLISECONDS",
            DimensionMappings: [
                {
                    SourceColumn: "registration",
                    DestinationColumn: "license"
                }
            ],
            MultiMeasureMappings: {
```

```
                TargetMultiMeasureName: "mva_measure_name",
                MultiMeasureAttributeMappings: [
                    {
                        SourceColumn: "wgt",
                        TargetMultiMeasureAttributeName: "weight",
                        MeasureValueType: "DOUBLE"
                    },
                    {
                        SourceColumn: "spd",
                        TargetMultiMeasureAttributeName: "speed",
                        MeasureValueType: "DOUBLE"
                    },
                    {
                        SourceColumn: "fuel_consumption",
                        TargetMultiMeasureAttributeName: "fuel",
                        MeasureValueType: "DOUBLE"
                    }
                ]
            }
        }
    },
 DataSourceConfiguration: {
        DataSourceS3Configuration: {
            BucketName: "test-batch-load-west-2",
            ObjectKeyPrefix: "sample.csv"
        },
        DataFormat: "CSV"
    },
    ReportConfiguration: {
        ReportS3Configuration: {
            BucketName: "test-batch-load-report-west-2",
            EncryptionOption: "SSE_S3"
        }
    }
};

const command = new CreateBatchLoadTaskCommand(params);

try {
    const data = await writeClient.send(command);
    console.log(`Created batch load task ` + data.TaskId);
} catch (error) {
 console.log("Error creating table. ", error);
 throw error;
```

```
}
```

.NET

```
using System;
using System.IO;
using System.Collections.Generic;
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
using System.Threading.Tasks;

namespace TimestreamDotNetSample
{
    public class CreateBatchLoadTaskExample
    {
        public const string DATABASE_NAME = "<database name>";
        public const string TABLE_NAME = "<table name>";
        public const string INPUT_BUCKET = "<input bucket name>";
        public const string INPUT_OBJECT_KEY_PREFIX = "<CSV file name>";
        public const string REPORT_BUCKET = "<report bucket name>";
        public const long HT_TTL_HOURS = 24L;
        public const long CT_TTL_DAYS = 7L;
        private readonly AmazonTimestreamWriteClient writeClient;

        public CreateBatchLoadTaskExample(AmazonTimestreamWriteClient writeClient)
        {
            this.writeClient = writeClient;
        }

        public async Task CreateBatchLoadTask()
        {
            try
            {
                var createBatchLoadTaskRequest = new CreateBatchLoadTaskRequest
                {
                    DataModelConfiguration = new DataModelConfiguration
                    {
                        DataModel = new DataModel
                        {
                            TimeColumn = "timestamp",
                            TimeUnit = TimeUnit.SECONDS,
                            DimensionMappings = new List<DimensionMapping>()
                            {
```

```
                                new()
                                {
                                        SourceColumn = "vehicle"
                                },
                                new()
                                {
                                        SourceColumn = "registration",
                                        DestinationColumn = "license"
                                }
                            },
                            MultiMeasureMappings = new MultiMeasureMappings
                            {
                                TargetMultiMeasureName = "mva_measure_name",
                                MultiMeasureAttributeMappings = new
List<MultiMeasureAttributeMapping>()
                                {
                                        new()
                                        {
                                                SourceColumn = "wgt",
                                                TargetMultiMeasureAttributeName =
"weight",
                                                MeasureValueType =
ScalarMeasureValueType.DOUBLE
                                        },
                                        new()
                                        {
                                                SourceColumn = "spd",
                                                TargetMultiMeasureAttributeName =
"speed",
                                                MeasureValueType =
ScalarMeasureValueType.DOUBLE
                                        },
                                        new()
                                        {
                                                SourceColumn = "fuel",
                                                TargetMultiMeasureAttributeName =
"fuel",
                                                MeasureValueType =
ScalarMeasureValueType.DOUBLE
                                        },
                                        new()
                                        {
                                                SourceColumn = "miles",
```

```
                                                    TargetMultiMeasureAttributeName =
 "miles",
                                                    MeasureValueType =
 ScalarMeasureValueType.DOUBLE
                                        }
                                    }
                                }
                            }
                        },
                        DataSourceConfiguration = new DataSourceConfiguration
                        {
                            DataSourceS3Configuration = new DataSourceS3Configuration
                            {
                                BucketName = INPUT_BUCKET,
                                ObjectKeyPrefix = INPUT_OBJECT_KEY_PREFIX
                            },
                            DataFormat = "CSV"
                        },
                        ReportConfiguration = new ReportConfiguration
                        {
                            ReportS3Configuration = new ReportS3Configuration
                            {
                                BucketName = REPORT_BUCKET
                            }
                        },
                        TargetDatabaseName = DATABASE_NAME,
                        TargetTableName = TABLE_NAME
                    };

                    CreateBatchLoadTaskResponse response = await
 writeClient.CreateBatchLoadTaskAsync(createBatchLoadTaskRequest);
                    Console.WriteLine($"Task created: " + response.TaskId);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Create batch load task failed:" + e.ToString());
                }
            }
        }
}
```

```
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
```

```
using Amazon;
using Amazon.TimestreamQuery;
using System.Threading.Tasks;
using System;
using CommandLine;
static class Constants
{

}
namespace TimestreamDotNetSample
{
    class MainClass
    {
        public class Options
        {

        }
        public static void Main(string[] args)
        {
            Parser.Default.ParseArguments<Options>(args)
                .WithParsed<Options>(o => {
                    MainAsync().GetAwaiter().GetResult();
                });
        }

        static async Task MainAsync()
        {
            var writeClientConfig = new AmazonTimestreamWriteConfig
            {
                ServiceURL =  "<service URL>",
                Timeout = TimeSpan.FromSeconds(20),
                MaxErrorRetry = 10
            };

            var writeClient = new AmazonTimestreamWriteClient(writeClientConfig);
            var example = new CreateBatchLoadTaskExample(writeClient);
            await example.CreateBatchLoadTask();
        }
    }
}
```

Describe batch load task

You can use the following code snippets to describe batch load tasks.

Java

```
    public void describeBatchLoadTask(String taskId) {
            final DescribeBatchLoadTaskResponse batchLoadTaskResponse =
 amazonTimestreamWrite

 .describeBatchLoadTask(DescribeBatchLoadTaskRequest.builder()
                                            .taskId(taskId)
                                            .build());

            System.out.println("Task id: " +
 batchLoadTaskResponse.batchLoadTaskDescription().taskId());
            System.out.println("Status: " +
 batchLoadTaskResponse.batchLoadTaskDescription().taskStatusAsString());
            System.out.println("Records processed: "
                            +
 batchLoadTaskResponse.batchLoadTaskDescription().progressReport().recordsProcessed());
    }
```

Go

```
package main

import (
 "fmt"
 "context"
 "log"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/timestreamwrite"
)

func main() {
 customResolver := aws.EndpointResolverWithOptionsFunc(func(service, region string,
 options ...interface{}) (aws.Endpoint, error) {
  if service == timestreamwrite.ServiceID && region == "us-west-2" {
      return aws.Endpoint{
          PartitionID:   "aws",
          URL:           <URL>,
```

```
          SigningRegion: "us-west-2",
      }, nil
  }
  return aws.Endpoint{}, &aws.EndpointNotFoundError{}
 })

 cfg, err := config.LoadDefaultConfig(context.TODO(),
 config.WithEndpointResolverWithOptions(customResolver), config.WithRegion("us-
west-2"))

 if err != nil {
    log.Fatalf("failed to load configuration, %v", err)
 }

 client := timestreamwrite.NewFromConfig(cfg)

 response, err := client.DescribeBatchLoadTask(context.TODO(),
 &timestreamwrite.DescribeBatchLoadTaskInput{
  TaskId: aws.String("<TaskId>"),
 })

 fmt.Println(aws.ToString(response.BatchLoadTaskDescription.TaskId))
}
```

Python

```
import boto3
from botocore.config import Config

INGEST_ENDPOINT="<url>"
REGION="us-west-2"
HT_TTL_HOURS = 24
CT_TTL_DAYS = 7
TASK_ID = "<task id>"

def describe_batch_load_task(client, task_id):
    try:
        result = client.describe_batch_load_task(TaskId=task_id)
        print("Successfully described batch load task: ", result)
    except Exception as err:
        print("Describe batch load task job failed:", err)
```

```
if __name__ == '__main__':
    session = boto3.Session()

    write_client = session.client('timestream-write', \
        endpoint_url=INGEST_ENDPOINT, region_name=REGION, \
        config=Config(read_timeout=20, max_pool_connections = 5000,
 retries={'max_attempts': 10}))

    describe_batch_load_task(write_client, TASK_ID)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

For API details, see Class DescribeBatchLoadCommand and DescribeBatchLoadTask.

```
import { TimestreamWriteClient, DescribeBatchLoadTaskCommand } from "@aws-sdk/
client-timestream-write";
const writeClient = new TimestreamWriteClient({ region: "<region>", endpoint:
 "<endpoint>" });

const params = {
    TaskId: "<TaskId>"
};

const command = new DescribeBatchLoadTaskCommand(params);

try {
    const data = await writeClient.send(command);
    console.log(`Batch load task has id ` + data.BatchLoadTaskDescription.TaskId);
} catch (error) {
    if (error.code === 'ResourceNotFoundException') {
        console.log("Batch load task doesn't exist.");
    } else {
        console.log("Describe batch load task failed.", error);
        throw error;
    }
}
```

.NET

```
using System;
```

```
using System.IO;
using System.Collections.Generic;
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
using System.Threading.Tasks;

namespace TimestreamDotNetSample
{
    public class DescribeBatchLoadTaskExample
    {
        private readonly AmazonTimestreamWriteClient writeClient;

        public DescribeBatchLoadTaskExample(AmazonTimestreamWriteClient writeClient)
        {
            this.writeClient = writeClient;
        }

        public async Task DescribeBatchLoadTask(String taskId)
        {
            try
            {
                var describeBatchLoadTaskRequest = new DescribeBatchLoadTaskRequest
                {
                    TaskId = taskId
                };
                DescribeBatchLoadTaskResponse response = await
 writeClient.DescribeBatchLoadTaskAsync(describeBatchLoadTaskRequest);
                Console.WriteLine($"Task has id:
{response.BatchLoadTaskDescription.TaskId}");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine("Batch load task does not exist.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Describe batch load task failed:" +
 e.ToString());
            }
        }
    }
}
```

```
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
using Amazon;
using Amazon.TimestreamQuery;
using System.Threading.Tasks;
using System;
using CommandLine;
static class Constants
{

}
namespace TimestreamDotNetSample
{
    class MainClass
    {
        public class Options
        {

        }
        public static void Main(string[] args)
        {
            Parser.Default.ParseArguments<Options>(args)
                .WithParsed<Options>(o => {
                    MainAsync().GetAwaiter().GetResult();
                });
        }

        static async Task MainAsync()
        {
            var writeClientConfig = new AmazonTimestreamWriteConfig
            {
                ServiceURL =  "<service URL>",
                Timeout = TimeSpan.FromSeconds(20),
                MaxErrorRetry = 10
            };

            var writeClient = new AmazonTimestreamWriteClient(writeClientConfig);
            var example = new DescribeBatchLoadTaskExample(writeClient);
            await example.DescribeBatchLoadTask("<batch load task id>");
        }
    }
}
```

List batch load tasks

You can use the following code snippets to list batch load tasks.

Java

```
    public void listBatchLoadTasks() {
            final ListBatchLoadTasksResponse listBatchLoadTasksResponse =
 amazonTimestreamWrite
                            .listBatchLoadTasks(ListBatchLoadTasksRequest.builder()
                                            .maxResults(15)
                                            .build());

            for (BatchLoadTask batchLoadTask :
 listBatchLoadTasksResponse.batchLoadTasks()) {
                    System.out.println(batchLoadTask.taskId());
            }
    }
```

Go

```
package main

import (
 "fmt"
 "context"
 "log"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/timestreamwrite"
)

func main() {
 customResolver := aws.EndpointResolverWithOptionsFunc(func(service, region string,
 options ...interface{}) (aws.Endpoint, error) {
  if service == timestreamwrite.ServiceID && region == "us-west-2" {
      return aws.Endpoint{
          PartitionID:   "aws",
          URL:           <URL>,
          SigningRegion: "us-west-2",
      }, nil
  }
  return aws.Endpoint{}, &aws.EndpointNotFoundError{}
```

```
 })

 cfg, err := config.LoadDefaultConfig(context.TODO(),
 config.WithEndpointResolverWithOptions(customResolver), config.WithRegion("us-
west-2"))

 if err != nil {
    log.Fatalf("failed to load configuration, %v", err)
 }

 client := timestreamwrite.NewFromConfig(cfg)
 listBatchLoadTasksMaxResult := int32(15)

 response, err := client.ListBatchLoadTasks(context.TODO(),
 &timestreamwrite.ListBatchLoadTasksInput{
  MaxResults: &listBatchLoadTasksMaxResult,
 })

 for i, task := range response.BatchLoadTasks {
  fmt.Println(i, aws.ToString(task.TaskId))
 }
}
```

Python

```
import boto3
from botocore.config import Config

INGEST_ENDPOINT = "<url>"
REGION = "us-west-2"
HT_TTL_HOURS = 24
CT_TTL_DAYS = 7


def print_batch_load_tasks(batch_load_tasks):
    for batch_load_task in batch_load_tasks:
        print(batch_load_task['TaskId'])


def list_batch_load_tasks(client):
    print("\nListing batch load tasks")
    try:
        response = client.list_batch_load_tasks(MaxResults=10)
```

```
        print_batch_load_tasks(response['BatchLoadTasks'])
        next_token = response.get('NextToken', None)
        while next_token:
            response = client.list_batch_load_tasks(
                NextToken=next_token, MaxResults=10)
            print_batch_load_tasks(response['BatchLoadTasks'])
            next_token = response.get('NextToken', None)
    except Exception as err:
        print("List batch load tasks failed:", err)
        raise err


if __name__ == '__main__':
    session = boto3.Session()

    write_client = session.client('timestream-write',
                                  endpoint_url=INGEST_ENDPOINT, region_name=REGION,
                                  config=Config(read_timeout=20,
 max_pool_connections=5000, retries={'max_attempts': 10}))

    list_batch_load_tasks(write_client)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

For API details, see Class DescribeBatchLoadCommand and DescribeBatchLoadTask.

```
import { TimestreamWriteClient, ListBatchLoadTasksCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "<region>", endpoint:
 "<endpoint>" });

const params = {
    MaxResults: <15>
};

const command = new ListBatchLoadTasksCommand(params);

getBatchLoadTasksList(null);

async function getBatchLoadTasksList(nextToken) {
    if (nextToken) {
```

```
        params.NextToken = nextToken;
    }

    try {
        const data = await writeClient.send(command);

        data.BatchLoadTasks.forEach(function (task) {
            console.log(task.TaskId);
        });

        if (data.NextToken) {
            return getBatchLoadTasksList(data.NextToken);
        }
    } catch (error) {
        console.log("Error while listing batch load tasks", error);
    }
}
```

.NET

```
using System;
using System.IO;
using System.Collections.Generic;
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
using System.Threading.Tasks;

namespace TimestreamDotNetSample
{
    public class ListBatchLoadTasksExample
    {
        private readonly AmazonTimestreamWriteClient writeClient;

        public ListBatchLoadTasksExample(AmazonTimestreamWriteClient writeClient)
        {
            this.writeClient = writeClient;
        }

        public async Task ListBatchLoadTasks()
        {
            Console.WriteLine("Listing batch load tasks");

            try
```

```
            {
                var listBatchLoadTasksRequest = new ListBatchLoadTasksRequest
                {
                    MaxResults = 15
                };

                ListBatchLoadTasksResponse response = await
 writeClient.ListBatchLoadTasksAsync(listBatchLoadTasksRequest);

                PrintBatchLoadTasks(response.BatchLoadTasks);
                var nextToken = response.NextToken;

                while (nextToken != null)
                {
                    listBatchLoadTasksRequest.NextToken = nextToken;
                    response = await
 writeClient.ListBatchLoadTasksAsync(listBatchLoadTasksRequest);
                    PrintBatchLoadTasks(response.BatchLoadTasks);
                    nextToken = response.NextToken;
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("List batch load tasks failed:" + e.ToString());
            }
        }

        private void PrintBatchLoadTasks(List<BatchLoadTask> tasks)
        {
            foreach (BatchLoadTask task in tasks)
                Console.WriteLine($"Task:{task.TaskId}");
        }
    }
}
```

```
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
using Amazon;
using Amazon.TimestreamQuery;
using System.Threading.Tasks;
using System;
using CommandLine;
static class Constants
```

```
{

}
namespace TimestreamDotNetSample
{
    class MainClass
    {
        public class Options
        {

        }
        public static void Main(string[] args)
        {
            Parser.Default.ParseArguments<Options>(args)
                .WithParsed<Options>(o => {
                    MainAsync().GetAwaiter().GetResult();
                });
        }

        static async Task MainAsync()
        {
            var writeClientConfig = new AmazonTimestreamWriteConfig
            {
                ServiceURL =  "<service URL>",
                Timeout = TimeSpan.FromSeconds(20),
                MaxErrorRetry = 10
            };

            var writeClient = new AmazonTimestreamWriteClient(writeClientConfig);
            var example = new ListBatchLoadTasksExample(writeClient);
            await example.ListBatchLoadTasks();
        }
    }
}
```

Resume batch load task

You can use the following code snippets to resume batch load tasks.

Java

```
    public void resumeBatchLoadTask(String taskId) {
```

```
            try {
                    amazonTimestreamWrite

 .resumeBatchLoadTask(ResumeBatchLoadTaskRequest.builder()
                                                    .taskId(taskId)
                                                    .build());

                    System.out.println("Successfully resumed batch load task.");
            } catch (ValidationException validationException) {
                    System.out.println(validationException.getMessage());
            }
    }
```

Go

```
package main

import (
 "fmt"
 "context"
 "log"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/timestreamwrite"
)

func main() {
 customResolver := aws.EndpointResolverWithOptionsFunc(func(service, region string,
 options ...interface{}) (aws.Endpoint, error) {
  if service == timestreamwrite.ServiceID && region == "us-west-2" {
      return aws.Endpoint{
          PartitionID:   "aws",
          URL:           <URL>,
          SigningRegion: "us-west-2",
      }, nil
  }
  return aws.Endpoint{}, &aws.EndpointNotFoundError{}
 })

 cfg, err := config.LoadDefaultConfig(context.TODO(),
 config.WithEndpointResolverWithOptions(customResolver), config.WithRegion("us-
west-2"))
```

```
 if err != nil {
    log.Fatalf("failed to load configuration, %v", err)
 }

 client := timestreamwrite.NewFromConfig(cfg)

 response, err := client.ResumeBatchLoadTask(context.TODO(),
 &timestreamwrite.ResumeBatchLoadTaskInput{
  TaskId: aws.String("TaskId"),
 })

 if err != nil {
  fmt.Println("Error:")
  fmt.Println(err)
 } else {
  fmt.Println("Resume batch load task is successful")
  fmt.Println(response)
 }
}
```

Python

```
import boto3
from botocore.config import Config

INGEST_ENDPOINT="<url>"
REGION="us-west-2"
HT_TTL_HOURS = 24
CT_TTL_DAYS = 7
TASK_ID = "<TaskId>"

def resume_batch_load_task(client, task_id):
    try:
        result = client.resume_batch_load_task(TaskId=task_id)
        print("Successfully resumed batch load task: ", result)
    except Exception as err:
        print("Resume batch load task failed:", err)


if __name__ == '__main__':
    session = boto3.Session()

    write_client = session.client('timestream-write', \
```

```
        endpoint_url=INGEST_ENDPOINT, region_name=REGION, \
        config=Config(read_timeout=20, max_pool_connections = 5000,
 retries={'max_attempts': 10}))

    resume_batch_load_task(write_client, TASK_ID)
```

Node.js

The following snippet uses AWS SDK for JavaScript v3. For more information about how to install the client and usage, see Timestream Write Client - AWS SDK for JavaScript v3.

For API details, see Class CreateBatchLoadCommand and CreateBatchLoadTask.

```
import { TimestreamWriteClient, ResumeBatchLoadTaskCommand } from "@aws-sdk/client-
timestream-write";
const writeClient = new TimestreamWriteClient({ region: "<region>", endpoint:
 "<endpoint>" });

const params = {
    TaskId: "<TaskId>"
};

const command = new ResumeBatchLoadTaskCommand(params);

try {
    const data = await writeClient.send(command);
    console.log("Resumed batch load task");
} catch (error) {
    console.log("Resume batch load task failed.", error);
 throw error;
}
```

.NET

```
using System;
using System.IO;
using System.Collections.Generic;
using Amazon.TimestreamWrite;
using Amazon.TimestreamWrite.Model;
using System.Threading.Tasks;

namespace TimestreamDotNetSample
{
```

```
    public class ResumeBatchLoadTaskExample
    {
        private readonly AmazonTimestreamWriteClient writeClient;

        public ResumeBatchLoadTaskExample(AmazonTimestreamWriteClient writeClient)
        {
            this.writeClient = writeClient;
        }

        public async Task ResumeBatchLoadTask(String taskId)
        {
            try
            {
                var resumeBatchLoadTaskRequest = new ResumeBatchLoadTaskRequest
                {
                    TaskId = taskId
                };
                ResumeBatchLoadTaskResponse response = await
 writeClient.ResumeBatchLoadTaskAsync(resumeBatchLoadTaskRequest);
                Console.WriteLine("Successfully resumed batch load task.");
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine("Batch load task does not exist.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Resume batch load task failed: " + e.ToString());
            }
        }
    }
}
```

Create scheduled query

You can use the following code snippets to create a scheduled query with multi-measure mapping.

Java

```
public static String DATABASE_NAME = "devops_sample_application";
public static String TABLE_NAME = "host_metrics_sample_application";
public static String HOSTNAME = "host-24Gju";
```

```
public static String SQ_NAME = "daily-sample";
public static String SCHEDULE_EXPRESSION = "cron(0/2 * * * ? *)";

// Find the average, p90, p95, and p99 CPU utilization for a specific EC2 host over
 the past 2 hours.
public static String QUERY = "SELECT region, az, hostname, BIN(time, 15s) AS
 binned_timestamp, " +
"ROUND(AVG(cpu_utilization), 2) AS avg_cpu_utilization, " +
"ROUND(APPROX_PERCENTILE(cpu_utilization, 0.9), 2) AS p90_cpu_utilization, " +
"ROUND(APPROX_PERCENTILE(cpu_utilization, 0.95), 2) AS p95_cpu_utilization, " +
"ROUND(APPROX_PERCENTILE(cpu_utilization, 0.99), 2) AS p99_cpu_utilization " +
"FROM " +  DATABASE_NAME + "." +  TABLE_NAME + " " +
"WHERE measure_name = 'metrics' " +
"AND hostname = '" + HOSTNAME + "' " +
"AND time > ago(2h) " +
"GROUP BY region, hostname, az, BIN(time, 15s) " +
"ORDER BY binned_timestamp ASC " +
"LIMIT 5";


public String createScheduledQuery(String topic_arn,
    String role_arn,
    String database_name,
    String table_name) {
    System.out.println("Creating Scheduled Query");

    List<Pair<String, MeasureValueType>> sourceColToMeasureValueTypes =
 Arrays.asList(
        Pair.of("avg_cpu_utilization", DOUBLE),
        Pair.of("p90_cpu_utilization", DOUBLE),
        Pair.of("p95_cpu_utilization", DOUBLE),
        Pair.of("p99_cpu_utilization", DOUBLE));

    CreateScheduledQueryRequest createScheduledQueryRequest = new
 CreateScheduledQueryRequest()
            .withName(SQ_NAME)
            .withQueryString(QUERY)
            .withScheduleConfiguration(new ScheduleConfiguration()
                    .withScheduleExpression(SCHEDULE_EXPRESSION))
            .withNotificationConfiguration(new NotificationConfiguration()
                    .withSnsConfiguration(new SnsConfiguration()
                            .withTopicArn(topic_arn)))
            .withTargetConfiguration(new
 TargetConfiguration().withTimestreamConfiguration(new TimestreamConfiguration()
```

```
                    .withDatabaseName(database_name)
                    .withTableName(table_name)
                    .withTimeColumn("binned_timestamp")
                    .withDimensionMappings(Arrays.asList(
                            new DimensionMapping()
                                    .withName("region")
                                    .withDimensionValueType("VARCHAR"),
                            new DimensionMapping()
                                    .withName("az")
                                    .withDimensionValueType("VARCHAR"),
                            new DimensionMapping()
                                    .withName("hostname")
                                    .withDimensionValueType("VARCHAR")
                    ))
                    .withMultiMeasureMappings(new MultiMeasureMappings()
                        .withTargetMultiMeasureName("multi-metrics")
                        .withMultiMeasureAttributeMappings(
                            sourceColToMeasureValueTypes.stream()
                            .map(pair -> new MultiMeasureAttributeMapping()
                                .withMeasureValueType(pair.getValue().name())
                                .withSourceColumn(pair.getKey()))
                            .collect(Collectors.toList()))))
            .withErrorReportConfiguration(new ErrorReportConfiguration()
                    .withS3Configuration(new S3Configuration()

 .withBucketName(timestreamDependencyHelper.getS3ErrorReportBucketName())))
            .withScheduledQueryExecutionRoleArn(role_arn);

    try {
        final CreateScheduledQueryResult createScheduledQueryResult =
 queryClient.createScheduledQuery(createScheduledQueryRequest);
        final String scheduledQueryArn = createScheduledQueryResult.getArn();
        System.out.println("Successfully created scheduled query : " +
 scheduledQueryArn);
        return scheduledQueryArn;
    }
    catch (Exception e) {
        System.out.println("Scheduled Query creation failed: " + e);
        throw e;
    }
}
```

Java v2

```
public static String DATABASE_NAME = "testJavaV2DB";
public static String TABLE_NAME = "testJavaV2Table";
public static String HOSTNAME = "host-24Gju";
public static String SQ_NAME = "daily-sample";
public static String SCHEDULE_EXPRESSION = "cron(0/2 * * * ? *)";

// Find the average, p90, p95, and p99 CPU utilization for a specific EC2 host over
 the past 2 hours.
public static String VALID_QUERY = "SELECT region, az, hostname, BIN(time, 15s) AS
 binned_timestamp, " +
"ROUND(AVG(cpu_utilization), 2) AS avg_cpu_utilization, " +
"ROUND(APPROX_PERCENTILE(cpu_utilization, 0.9), 2) AS p90_cpu_utilization, " +
"ROUND(APPROX_PERCENTILE(cpu_utilization, 0.95), 2) AS p95_cpu_utilization, " +
"ROUND(APPROX_PERCENTILE(cpu_utilization, 0.99), 2) AS p99_cpu_utilization " +
"FROM " +  DATABASE_NAME + "." +  TABLE_NAME + " " +
"WHERE measure_name = 'metrics' " +
"AND hostname = '" + HOSTNAME + "' " +
"AND time > ago(2h) " +
"GROUP BY region, hostname, az, BIN(time, 15s) " +
"ORDER BY binned_timestamp ASC " +
"LIMIT 5";


private String createScheduledQueryHelper(String topicArn, String roleArn,
        String s3ErrorReportBucketName, String query,
        TargetConfiguration targetConfiguration) {
    System.out.println("Creating Scheduled Query");

    CreateScheduledQueryRequest createScheduledQueryRequest =
 CreateScheduledQueryRequest.builder()
            .name(SQ_NAME)
            .queryString(query)
            .scheduleConfiguration(ScheduleConfiguration.builder()
                    .scheduleExpression(SCHEDULE_EXPRESSION)
                    .build())
            .notificationConfiguration(NotificationConfiguration.builder()
                    .snsConfiguration(SnsConfiguration.builder()
                            .topicArn(topicArn)
                            .build())
                    .build())
            .targetConfiguration(targetConfiguration)
            .errorReportConfiguration(ErrorReportConfiguration.builder()
```

```
                    .s3Configuration(S3Configuration.builder()
                            .bucketName(s3ErrorReportBucketName)
                            .objectKeyPrefix(SCHEDULED_QUERY_EXAMPLE)
                            .build())
                    .build())
            .scheduledQueryExecutionRoleArn(roleArn)
            .build();

    try {
        final CreateScheduledQueryResponse response =
 queryClient.createScheduledQuery(createScheduledQueryRequest);
        final String scheduledQueryArn = response.arn();
        System.out.println("Successfully created scheduled query : " +
 scheduledQueryArn);
        return scheduledQueryArn;
    }
    catch (Exception e) {
        System.out.println("Scheduled Query creation failed: " + e);
        throw e;
    }
}

public String createScheduledQuery(String topicArn, String roleArn,
        String databaseName, String tableName, String s3ErrorReportBucketName) {
    List<Pair<String, MeasureValueType>> sourceColToMeasureValueTypes =
 Arrays.asList(
            Pair.of("avg_cpu_utilization", DOUBLE),
            Pair.of("p90_cpu_utilization", DOUBLE),
            Pair.of("p95_cpu_utilization", DOUBLE),
            Pair.of("p99_cpu_utilization", DOUBLE));

    TargetConfiguration targetConfiguration = TargetConfiguration.builder()
            .timestreamConfiguration(TimestreamConfiguration.builder()
            .databaseName(databaseName)
            .tableName(tableName)
            .timeColumn("binned_timestamp")
            .dimensionMappings(Arrays.asList(
                    DimensionMapping.builder()
                            .name("region")
                            .dimensionValueType("VARCHAR")
                            .build(),
                    DimensionMapping.builder()
                            .name("az")
                            .dimensionValueType("VARCHAR")
```

```
                            .build(),
                    DimensionMapping.builder()
                            .name("hostname")
                            .dimensionValueType("VARCHAR")
                            .build()
            ))
            .multiMeasureMappings(MultiMeasureMappings.builder()
                    .targetMultiMeasureName("multi-metrics")
                    .multiMeasureAttributeMappings(
                            sourceColToMeasureValueTypes.stream()
                                    .map(pair ->
 MultiMeasureAttributeMapping.builder()

 .measureValueType(pair.getValue().name())
                                            .sourceColumn(pair.getKey())
                                            .build())
                                    .collect(Collectors.toList()))
                    .build())
            .build())
            .build();

    return createScheduledQueryHelper(topicArn, roleArn, s3ErrorReportBucketName,
 VALID_QUERY, targetConfiguration);
}}
```

Go

```
SQ_ERROR_CONFIGURATION_S3_BUCKET_NAME_PREFIX = "sq-error-configuration-sample-s3-
bucket-"
HOSTNAME            = "host-24Gju"
SQ_NAME             = "daily-sample"
SCHEDULE_EXPRESSION = "cron(0/1 * * * ? *)"
QUERY               = "SELECT region, az, hostname, BIN(time, 15s) AS
 binned_timestamp, " +
    "ROUND(AVG(cpu_utilization), 2) AS avg_cpu_utilization, " +
    "ROUND(APPROX_PERCENTILE(cpu_utilization, 0.9), 2) AS p90_cpu_utilization, " +
    "ROUND(APPROX_PERCENTILE(cpu_utilization, 0.95), 2) AS p95_cpu_utilization, " +
    "ROUND(APPROX_PERCENTILE(cpu_utilization, 0.99), 2) AS p99_cpu_utilization " +
    "FROM %s.%s " +
    "WHERE measure_name = 'metrics' " +
    "AND hostname = '" + HOSTNAME + "' " +
    "AND time > ago(2h) " +
    "GROUP BY region, hostname, az, BIN(time, 15s) " +
```

```
    "ORDER BY binned_timestamp ASC " +
    "LIMIT 5"
s3BucketName = utils.SQ_ERROR_CONFIGURATION_S3_BUCKET_NAME_PREFIX +
 generateRandomStringWithSize(5)

func generateRandomStringWithSize(size int) string {
     rand.Seed(time.Now().UnixNano())
     alphaNumericList := []rune("abcdefghijklmnopqrstuvwxyz0123456789")
     randomPrefix := make([]rune, size)
     for i := range randomPrefix {
         randomPrefix[i] = alphaNumericList[rand.Intn(len(alphaNumericList))]
     }
     return string(randomPrefix)
 }

func (timestreamBuilder TimestreamBuilder) createScheduledQuery(topicArn string,
 roleArn string, s3ErrorReportBucketName string,
query string, targetConfiguration timestreamquery.TargetConfiguration) (string,
 error) {

createScheduledQueryInput := &timestreamquery.CreateScheduledQueryInput{
    Name:        aws.String(SQ_NAME),
    QueryString: aws.String(query),
    ScheduleConfiguration: &timestreamquery.ScheduleConfiguration{
        ScheduleExpression: aws.String(SCHEDULE_EXPRESSION),
    },
    NotificationConfiguration: &timestreamquery.NotificationConfiguration{
        SnsConfiguration: &timestreamquery.SnsConfiguration{
            TopicArn: aws.String(topicArn),
        },
    },
    TargetConfiguration: &targetConfiguration,
    ErrorReportConfiguration: &timestreamquery.ErrorReportConfiguration{
        S3Configuration: &timestreamquery.S3Configuration{
            BucketName: aws.String(s3ErrorReportBucketName),
        },
    },
    ScheduledQueryExecutionRoleArn: aws.String(roleArn),
}

createScheduledQueryOutput, err :=
 timestreamBuilder.QuerySvc.CreateScheduledQuery(createScheduledQueryInput)

if err != nil {
```

```
    fmt.Printf("Error: %s", err.Error())
} else {
    fmt.Println("createScheduledQueryResult is successful")
    return *createScheduledQueryOutput.Arn, nil
}
return "", err
}

 func (timestreamBuilder TimestreamBuilder) CreateValidScheduledQuery(topicArn
 string, roleArn string, s3ErrorReportBucketName string,
     sqDatabaseName string, sqTableName string, databaseName string, tableName
 string) (string, error) {

     targetConfiguration := timestreamquery.TargetConfiguration{
         TimestreamConfiguration: &timestreamquery.TimestreamConfiguration{
             DatabaseName: aws.String(sqDatabaseName),
             TableName:    aws.String(sqTableName),
             TimeColumn:   aws.String("binned_timestamp"),
             DimensionMappings: []*timestreamquery.DimensionMapping{
                 {
                     Name:               aws.String("region"),
                     DimensionValueType: aws.String("VARCHAR"),
                 },
                 {
                     Name:               aws.String("az"),
                     DimensionValueType: aws.String("VARCHAR"),
                 },
                 {
                     Name:               aws.String("hostname"),
                     DimensionValueType: aws.String("VARCHAR"),
                 },
             },
             MultiMeasureMappings: &timestreamquery.MultiMeasureMappings{
                 TargetMultiMeasureName: aws.String("multi-metrics"),
                 MultiMeasureAttributeMappings:
 []*timestreamquery.MultiMeasureAttributeMapping{
                     {
                         SourceColumn:     aws.String("avg_cpu_utilization"),
                         MeasureValueType:
 aws.String(timestreamquery.MeasureValueTypeDouble),
                     },
                     {
                         SourceColumn:     aws.String("p90_cpu_utilization"),
```

```
                        MeasureValueType:
aws.String(timestreamquery.MeasureValueTypeDouble),
                    },
                    {
                        SourceColumn:     aws.String("p95_cpu_utilization"),
                        MeasureValueType:
aws.String(timestreamquery.MeasureValueTypeDouble),
                    },
                    {
                        SourceColumn:     aws.String("p99_cpu_utilization"),
                        MeasureValueType:
aws.String(timestreamquery.MeasureValueTypeDouble),
                    },
                },
            },
        },
    }
    return timestreamBuilder.createScheduledQuery(topicArn, roleArn,
s3ErrorReportBucketName,
        fmt.Sprintf(QUERY, databaseName, tableName), targetConfiguration)
}
```

Python

```
HOSTNAME = "host-24Gju"
SQ_NAME = "daily-sample"
ERROR_BUCKET_NAME = "scheduledquerysamplerrorbucket" +
 ''.join([choice(ascii_lowercase) for _ in range(5)])
QUERY = \
    "SELECT region, az, hostname, BIN(time, 15s) AS binned_timestamp, " \
    "    ROUND(AVG(cpu_utilization), 2) AS avg_cpu_utilization, " \
    "    ROUND(APPROX_PERCENTILE(cpu_utilization, 0.9), 2) AS p90_cpu_utilization, "
 \
    "    ROUND(APPROX_PERCENTILE(cpu_utilization, 0.95), 2) AS p95_cpu_utilization,
 " \
    "    ROUND(APPROX_PERCENTILE(cpu_utilization, 0.99), 2) AS p99_cpu_utilization "
 \
    "FROM " + database_name + "." + table_name + " " \
    "WHERE measure_name = 'metrics' " \
    "AND hostname = '" + self.HOSTNAME + "' " \
    "AND time > ago(2h) " \
    "GROUP BY region, hostname, az, BIN(time, 15s) " \
    "ORDER BY binned_timestamp ASC " \
```

```
    "LIMIT 5"

def create_scheduled_query_helper(self, topic_arn, role_arn, query,
 target_configuration):
    print("\nCreating Scheduled Query")
    schedule_configuration = {
        'ScheduleExpression': 'cron(0/2 * * * ? *)'
    }
    notification_configuration = {
        'SnsConfiguration': {
            'TopicArn': topic_arn
        }
    }
    error_report_configuration = {
        'S3Configuration': {
            'BucketName': ERROR_BUCKET_NAME
        }
    }

    try:
        create_scheduled_query_response = \
            query_client.create_scheduled_query(Name=self.SQ_NAME,
                 QueryString=query,
                 ScheduleConfiguration=schedule_configuration,
                 NotificationConfiguration=notification_configuration,
                 TargetConfiguration=target_configuration,
                 ScheduledQueryExecutionRoleArn=role_arn,
                 ErrorReportConfiguration=error_report_configuration
                 )
        print("Successfully created scheduled query : ",
 create_scheduled_query_response['Arn'])
        return create_scheduled_query_response['Arn']
    except Exception as err:
        print("Scheduled Query creation failed:", err)
        raise err

def create_valid_scheduled_query(self, topic_arn, role_arn):
    target_configuration = {
        'TimestreamConfiguration': {
            'DatabaseName': self.sq_database_name,
            'TableName': self.sq_table_name,
            'TimeColumn': 'binned_timestamp',
            'DimensionMappings': [
                {'Name': 'region', 'DimensionValueType': 'VARCHAR'},
```

```
                {'Name': 'az', 'DimensionValueType': 'VARCHAR'},
                {'Name': 'hostname', 'DimensionValueType': 'VARCHAR'}
            ],
            'MultiMeasureMappings': {
                'TargetMultiMeasureName': 'target_name',
                'MultiMeasureAttributeMappings': [
                    {'SourceColumn': 'avg_cpu_utilization', 'MeasureValueType':
 'DOUBLE',
                     'TargetMultiMeasureAttributeName': 'avg_cpu_utilization'},
                    {'SourceColumn': 'p90_cpu_utilization', 'MeasureValueType':
 'DOUBLE',
                     'TargetMultiMeasureAttributeName': 'p90_cpu_utilization'},
                    {'SourceColumn': 'p95_cpu_utilization', 'MeasureValueType':
 'DOUBLE',
                     'TargetMultiMeasureAttributeName': 'p95_cpu_utilization'},
                    {'SourceColumn': 'p99_cpu_utilization', 'MeasureValueType':
 'DOUBLE',
                     'TargetMultiMeasureAttributeName': 'p99_cpu_utilization'},
                ]
            }
        }
    }

    return self.create_scheduled_query_helper(topic_arn, role_arn, QUERY,
 target_configuration)
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
const DATABASE_NAME = 'devops_sample_application';
const TABLE_NAME = 'host_metrics_sample_application';
const SQ_DATABASE_NAME = 'sq_result_database';
const SQ_TABLE_NAME = 'sq_result_table';
const HOSTNAME = "host-24Gju";
const SQ_NAME = "daily-sample";
const SCHEDULE_EXPRESSION = "cron(0/1 * * * ? *)";

// Find the average, p90, p95, and p99 CPU utilization for a specific EC2 host over
 the past 2 hours.
const VALID_QUERY = "SELECT region, az, hostname, BIN(time, 15s) AS
 binned_timestamp, " +
```

```
    " ROUND(AVG(cpu_utilization), 2) AS avg_cpu_utilization, " +
    " ROUND(APPROX_PERCENTILE(cpu_utilization, 0.9), 2) AS p90_cpu_utilization, " +
    " ROUND(APPROX_PERCENTILE(cpu_utilization, 0.95), 2) AS p95_cpu_utilization, " +
    " ROUND(APPROX_PERCENTILE(cpu_utilization, 0.99), 2) AS p99_cpu_utilization " +
    "FROM " + DATABASE_NAME + "." + TABLE_NAME + " " +
    "WHERE measure_name = 'metrics' " +
    " AND hostname = '" + HOSTNAME + "' " +
    " AND time > ago(2h) " +
    "GROUP BY region, hostname, az, BIN(time, 15s) " +
    "ORDER BY binned_timestamp ASC " +
    "LIMIT 5";

async function createScheduledQuery(topicArn, roleArn, s3ErrorReportBucketName) {
    console.log("Creating Valid Scheduled Query");
    const DimensionMappingList = [{
            'Name': 'region',
            'DimensionValueType': 'VARCHAR'
        },
        {
            'Name': 'az',
            'DimensionValueType': 'VARCHAR'
        },
        {
            'Name': 'hostname',
            'DimensionValueType': 'VARCHAR'
        }
    ];

    const MultiMeasureMappings = {
        TargetMultiMeasureName: "multi-metrics",
        MultiMeasureAttributeMappings: [{
                'SourceColumn': 'avg_cpu_utilization',
                'MeasureValueType': 'DOUBLE'
            },
            {
                'SourceColumn': 'p90_cpu_utilization',
                'MeasureValueType': 'DOUBLE'
            },
            {
                'SourceColumn': 'p95_cpu_utilization',
                'MeasureValueType': 'DOUBLE'
            },
            {
                'SourceColumn': 'p99_cpu_utilization',
```

```
                'MeasureValueType': 'DOUBLE'
            },
        ]
    }

    const timestreamConfiguration = {
        DatabaseName: SQ_DATABASE_NAME,
        TableName: SQ_TABLE_NAME,
        TimeColumn: "binned_timestamp",
        DimensionMappings: DimensionMappingList,
        MultiMeasureMappings: MultiMeasureMappings
    }

    const createScheduledQueryRequest = {
        Name: SQ_NAME,
        QueryString: VALID_QUERY,
        ScheduleConfiguration: {
            ScheduleExpression: SCHEDULE_EXPRESSION
        },
        NotificationConfiguration: {
            SnsConfiguration: {
                TopicArn: topicArn
            }
        },
        TargetConfiguration: {
            TimestreamConfiguration: timestreamConfiguration
        },
        ScheduledQueryExecutionRoleArn: roleArn,
        ErrorReportConfiguration: {
            S3Configuration: {
                BucketName: s3ErrorReportBucketName
            }
        }
    };
    try {
        const data = await
queryClient.createScheduledQuery(createScheduledQueryRequest).promise();
        console.log("Successfully created scheduled query: " + data.Arn);
        return data.Arn;
    } catch (err) {
        console.log("Scheduled Query creation failed: ", err);
        throw err;
    }
```

```
}
```

.NET

```
public const string Hostname = "host-24Gju";
public const string SqName = "timestream-sample";
public const string SqDatabaseName = "sq_result_database";
public const string SqTableName = "sq_result_table";

public const string ErrorConfigurationS3BucketNamePrefix = "error-configuration-
sample-s3-bucket-";
public const string ScheduleExpression = "cron(0/2 * * * ? *)";

// Find the average, p90, p95, and p99 CPU utilization for a specific EC2 host over
 the past 2 hours.
public const string ValidQuery = "SELECT region, az, hostname, BIN(time, 15s) AS
 binned_timestamp, " +
      "ROUND(AVG(cpu_utilization), 2) AS avg_cpu_utilization, " +
      "ROUND(APPROX_PERCENTILE(cpu_utilization, 0.9), 2) AS p90_cpu_utilization, " +
      "ROUND(APPROX_PERCENTILE(cpu_utilization, 0.95), 2) AS p95_cpu_utilization, "
 +
      "ROUND(APPROX_PERCENTILE(cpu_utilization, 0.99), 2) AS p99_cpu_utilization " +
      "FROM " + Constants.DATABASE_NAME + "." + Constants.TABLE_NAME + " " +
      "WHERE measure_name = 'metrics' " +
      "AND hostname = '" + Hostname + "' " +
      "AND time > ago(2h) " +
      "GROUP BY region, hostname, az, BIN(time, 15s) " +
      "ORDER BY binned_timestamp ASC " +
      "LIMIT 5";

private async Task<String> CreateValidScheduledQuery(string topicArn, string
 roleArn,
             string databaseName, string tableName, string s3ErrorReportBucketName)
 {
     List<MultiMeasureAttributeMapping> sourceColToMeasureValueTypes =
         new List<MultiMeasureAttributeMapping>()
         {
             new()
             {
                 SourceColumn = "avg_cpu_utilization",
                 MeasureValueType = MeasureValueType.DOUBLE.Value
             },
             new()
```

```
            {
                SourceColumn = "p90_cpu_utilization",
                MeasureValueType = MeasureValueType.DOUBLE.Value
            },
            new()
            {
                SourceColumn = "p95_cpu_utilization",
                MeasureValueType = MeasureValueType.DOUBLE.Value
            },
            new()
            {
                SourceColumn = "p99_cpu_utilization",
                MeasureValueType = MeasureValueType.DOUBLE.Value
            }
        };

    TargetConfiguration targetConfiguration = new TargetConfiguration()
    {
        TimestreamConfiguration = new TimestreamConfiguration()
        {
            DatabaseName = databaseName,
            TableName = tableName,
            TimeColumn = "binned_timestamp",
            DimensionMappings = new List<DimensionMapping>()
            {
                new()
                {
                    Name = "region",
                    DimensionValueType = "VARCHAR"
                },
                new()
                {
                    Name = "az",
                    DimensionValueType = "VARCHAR"
                },
                new()
                {
                    Name = "hostname",
                    DimensionValueType = "VARCHAR"
                }
            },
            MultiMeasureMappings = new MultiMeasureMappings()
            {
                TargetMultiMeasureName = "multi-metrics",
```

```
                MultiMeasureAttributeMappings = sourceColToMeasureValueTypes
            }
        }
    };
    return await CreateScheduledQuery(topicArn, roleArn, s3ErrorReportBucketName,
        ScheduledQueryConstants.ValidQuery, targetConfiguration);
}

private async Task<String> CreateScheduledQuery(string topicArn, string roleArn,
            string s3ErrorReportBucketName, string query, TargetConfiguration
 targetConfiguration)
 {
    try
    {
        Console.WriteLine("Creating Scheduled Query");
        CreateScheduledQueryResponse response = await
 _amazonTimestreamQuery.CreateScheduledQueryAsync(
            new CreateScheduledQueryRequest()
            {
                Name = ScheduledQueryConstants.SqName,
                QueryString = query,
                ScheduleConfiguration = new ScheduleConfiguration()
                {
                    ScheduleExpression = ScheduledQueryConstants.ScheduleExpression
                },
                NotificationConfiguration = new NotificationConfiguration()
                {
                    SnsConfiguration = new SnsConfiguration()
                    {
                        TopicArn = topicArn
                    }
                },
                TargetConfiguration = targetConfiguration,
                ErrorReportConfiguration = new ErrorReportConfiguration()
                {
                    S3Configuration = new S3Configuration()
                    {
                        BucketName = s3ErrorReportBucketName
                    }
                },
                ScheduledQueryExecutionRoleArn = roleArn
            });
        Console.WriteLine($"Successfully created scheduled query :
 {response.Arn}");
```

```
        return response.Arn;
    }
    catch (Exception e)
    {
        Console.WriteLine($"Scheduled Query creation failed: {e}");
        throw;
    }
}
```

List scheduled query

You can use the following code snippets to list your scheduled queries.

Java

```
public void listScheduledQueries() {
    System.out.println("Listing Scheduled Query");
    try {
        String nextToken = null;
        List<String> scheduledQueries = new ArrayList<>();

        do {
            ListScheduledQueriesResult listScheduledQueriesResult =
                    queryClient.listScheduledQueries(new
 ListScheduledQueriesRequest()
                            .withNextToken(nextToken).withMaxResults(10));
            List<ScheduledQuery> scheduledQueryList =
 listScheduledQueriesResult.getScheduledQueries();

            printScheduledQuery(scheduledQueryList);
            nextToken = listScheduledQueriesResult.getNextToken();
        } while (nextToken != null);
    }
    catch (Exception e) {
        System.out.println("List Scheduled Query failed: " + e);
        throw e;
    }
}

public void printScheduledQuery(List<ScheduledQuery> scheduledQueryList) {
    for (ScheduledQuery scheduledQuery: scheduledQueryList) {
        System.out.println(scheduledQuery.getArn());
```

```
    }
}
```

Java v2

```
public void listScheduledQueries() {
    System.out.println("Listing Scheduled Query");
    try {
        String nextToken = null;

        do {
            ListScheduledQueriesResponse listScheduledQueriesResult =

 queryClient.listScheduledQueries(ListScheduledQueriesRequest.builder()
                            .nextToken(nextToken).maxResults(10)
                            .build());
            List<ScheduledQuery> scheduledQueryList =
 listScheduledQueriesResult.scheduledQueries();

            printScheduledQuery(scheduledQueryList);
            nextToken = listScheduledQueriesResult.nextToken();
        } while (nextToken != null);
    }
    catch (Exception e) {
        System.out.println("List Scheduled Query failed: " + e);
        throw e;
    }
}

public void printScheduledQuery(List<ScheduledQuery> scheduledQueryList) {
    for (ScheduledQuery scheduledQuery: scheduledQueryList) {
        System.out.println(scheduledQuery.arn());
    }
}
```

Go

```
func (timestreamBuilder TimestreamBuilder) ListScheduledQueries()
 ([]*timestreamquery.ScheduledQuery, error) {

     var nextToken *string = nil
     var scheduledQueries []*timestreamquery.ScheduledQuery
     for ok := true; ok; ok = nextToken != nil {
```

```
        listScheduledQueriesInput := &timestreamquery.ListScheduledQueriesInput{
            MaxResults: aws.Int64(15),
        }
        if nextToken != nil {
            listScheduledQueriesInput.NextToken = aws.String(*nextToken)
        }

        listScheduledQueriesOutput, err :=
 timestreamBuilder.QuerySvc.ListScheduledQueries(listScheduledQueriesInput)
        if err != nil {
            fmt.Printf("Error: %s", err.Error())
            return nil, err
        }
        scheduledQueries = append(scheduledQueries,
 listScheduledQueriesOutput.ScheduledQueries...)
        nextToken = listScheduledQueriesOutput.NextToken
    }
    return scheduledQueries, nil
}
```

Python

```
def list_scheduled_queries(self):
    print("\nListing Scheduled Queries")
    try:
        response = self.query_client.list_scheduled_queries(MaxResults=10)
        self.print_scheduled_queries(response['ScheduledQueries'])
        next_token = response.get('NextToken', None)
        while next_token:
            response =
 self.query_client.list_scheduled_queries(NextToken=next_token, MaxResults=10)
            self.print_scheduled_queries(response['ScheduledQueries'])
            next_token = response.get('NextToken', None)
    except Exception as err:
        print("List scheduled queries failed:", err)
        raise err

@staticmethod
def print_scheduled_queries(scheduled_queries):
    for scheduled_query in scheduled_queries:
        print(scheduled_query['Arn'])
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function listScheduledQueries() {
     console.log("Listing Scheduled Query");
     try {
         var nextToken = null;
         do {
             var params = {
                 MaxResults: 10,
                 NextToken: nextToken
             }
             var data = await queryClient.listScheduledQueries(params).promise();
             var scheduledQueryList = data.ScheduledQueries;
             printScheduledQuery(scheduledQueryList);
             nextToken = data.NextToken;
         }
         while (nextToken != null);
     }  catch (err) {
         console.log("List Scheduled Query failed: ", err);
         throw err;
     }
 }

 async function printScheduledQuery(scheduledQueryList) {
     scheduledQueryList.forEach(element => console.log(element.Arn));
 }
```

.NET

```
private async Task ListScheduledQueries()
 {
     try
     {
         Console.WriteLine("Listing Scheduled Query");
         string nextToken;
         do
         {
             ListScheduledQueriesResponse response =
                 await _amazonTimestreamQuery.ListScheduledQueriesAsync(new
 ListScheduledQueriesRequest());
```

```
            foreach (var scheduledQuery in response.ScheduledQueries)
            {
                Console.WriteLine($"{scheduledQuery.Arn}");
            }

            nextToken = response.NextToken;
        } while (nextToken != null);
    }
    catch (Exception e)
    {
        Console.WriteLine($"List Scheduled Query failed: {e}");
        throw;
    }
}
```

Describe scheduled query

You can use the following code snippets to describe a scheduled query.

Java

```
public void describeScheduledQueries(String scheduledQueryArn) {
    System.out.println("Describing Scheduled Query");
    try {
        DescribeScheduledQueryResult describeScheduledQueryResult =
 queryClient.describeScheduledQuery(new
 DescribeScheduledQueryRequest().withScheduledQueryArn(scheduledQueryArn));
        System.out.println(describeScheduledQueryResult);
    }
    catch (ResourceNotFoundException e) {
        System.out.println("Scheduled Query doesn't exist");
        throw e;
    }
    catch (Exception e) {
        System.out.println("Describe Scheduled Query failed: " + e);
        throw e;
    }
}
```

Java v2

```
public void describeScheduledQueries(String scheduledQueryArn) {
```

```
    System.out.println("Describing Scheduled Query");
    try {
        DescribeScheduledQueryResponse describeScheduledQueryResult =

 queryClient.describeScheduledQuery(DescribeScheduledQueryRequest.builder()
                        .scheduledQueryArn(scheduledQueryArn)
                        .build());
        System.out.println(describeScheduledQueryResult);
    }
    catch (ResourceNotFoundException e) {
        System.out.println("Scheduled Query doesn't exist");
        throw e;
    }
    catch (Exception e) {
        System.out.println("Describe Scheduled Query failed: " + e);
        throw e;
    }
}
```

Go

```
func (timestreamBuilder TimestreamBuilder) DescribeScheduledQuery(scheduledQueryArn
 string) error {

     describeScheduledQueryInput := &timestreamquery.DescribeScheduledQueryInput{
         ScheduledQueryArn: aws.String(scheduledQueryArn),
     }
     describeScheduledQueryOutput, err :=
 timestreamBuilder.QuerySvc.DescribeScheduledQuery(describeScheduledQueryInput)

     if err != nil {
         if aerr, ok := err.(awserr.Error); ok {
             switch aerr.Code() {
             case timestreamquery.ErrCodeResourceNotFoundException:
                 fmt.Println(timestreamquery.ErrCodeResourceNotFoundException,
 aerr.Error())
             default:
                 fmt.Printf("Error: %s", err.Error())
             }
         } else {
             fmt.Printf("Error: %s", aerr.Error())
         }
         return err
```

```
    } else {
        fmt.Println("DescribeScheduledQuery is successful, below is the output:")
        fmt.Println(describeScheduledQueryOutput.ScheduledQuery)
        return nil
    }
}
```

Python

```
def describe_scheduled_query(self, scheduled_query_arn):
    print("\nDescribing Scheduled Query")
    try:
        response =
 self.query_client.describe_scheduled_query(ScheduledQueryArn=scheduled_query_arn)
        if 'ScheduledQuery' in response:
            response = response['ScheduledQuery']
            for key in response:
                print("{} :{}".format(key, response[key]))
    except self.query_client.exceptions.ResourceNotFoundException as err:
        print("Scheduled Query doesn't exist")
        raise err
    except Exception as err:
        print("Scheduled Query describe failed:", err)
        raise err
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at [Node.js sample Amazon Timestream for LiveAnalytics application on GitHub](#).

```
async function describeScheduledQuery(scheduledQueryArn) {
    console.log("Describing Scheduled Query");
    var params = {
        ScheduledQueryArn: scheduledQueryArn
    }
    try {
        const data = await queryClient.describeScheduledQuery(params).promise();
        console.log(data.ScheduledQuery);
    } catch (err) {
        console.log("Describe Scheduled Query failed: ", err);
        throw err;
    }
}
```

.NET

```
private async Task DescribeScheduledQuery(string scheduledQueryArn)
 {
     try
     {
         Console.WriteLine("Describing Scheduled Query");
         DescribeScheduledQueryResponse response = await
 _amazonTimestreamQuery.DescribeScheduledQueryAsync(
             new DescribeScheduledQueryRequest()
             {
                 ScheduledQueryArn = scheduledQueryArn
             });

 Console.WriteLine($"{JsonConvert.SerializeObject(response.ScheduledQuery)}");
     }
     catch (ResourceNotFoundException e)
     {
         Console.WriteLine($"Scheduled Query doesn't exist: {e}");
         throw;
     }
     catch (Exception e)
     {
         Console.WriteLine($"Describe Scheduled Query failed: {e}");
         throw;
     }
 }
```

Execute scheduled query

You can use the following code snippets to run a scheduled query.

Java

```
public void executeScheduledQueries(String scheduledQueryArn, Date invocationTime) {
    System.out.println("Executing Scheduled Query");
    try {
        ExecuteScheduledQueryResult executeScheduledQueryResult =
 queryClient.executeScheduledQuery(new ExecuteScheduledQueryRequest()
                .withScheduledQueryArn(scheduledQueryArn)
                .withInvocationTime(invocationTime)
        );
```

```
    }
    catch (ResourceNotFoundException e) {
        System.out.println("Scheduled Query doesn't exist");
        throw e;
    }
    catch (Exception e) {
        System.out.println("Execution Scheduled Query failed: " + e);
        throw e;
    }
}
```

Java v2

```
public void executeScheduledQuery(String scheduledQueryArn) {
    System.out.println("Executing Scheduled Query");
    try {
        ExecuteScheduledQueryResponse executeScheduledQueryResult =
 queryClient.executeScheduledQuery(ExecuteScheduledQueryRequest.builder()
                .scheduledQueryArn(scheduledQueryArn)
                .invocationTime(Instant.now())
                .build()
        );

        System.out.println("Execute ScheduledQuery response code: " +
 executeScheduledQueryResult.sdkHttpResponse().statusCode());

    }
    catch (ResourceNotFoundException e) {
        System.out.println("Scheduled Query doesn't exist");
        throw e;
    }
    catch (Exception e) {
        System.out.println("Execution Scheduled Query failed: " + e);
        throw e;
    }
}
```

Go

```
func (timestreamBuilder TimestreamBuilder) ExecuteScheduledQuery(scheduledQueryArn
 string, invocationTime time.Time) error {
```

```
    executeScheduledQueryInput := &timestreamquery.ExecuteScheduledQueryInput{
        ScheduledQueryArn: aws.String(scheduledQueryArn),
        InvocationTime:    aws.Time(invocationTime),
    }
    executeScheduledQueryOutput, err :=
 timestreamBuilder.QuerySvc.ExecuteScheduledQuery(executeScheduledQueryInput)

    if err != nil {
        if aerr, ok := err.(awserr.Error); ok {
            switch aerr.Code() {
            case timestreamquery.ErrCodeResourceNotFoundException:
                fmt.Println(timestreamquery.ErrCodeResourceNotFoundException,
 aerr.Error())
            default:
                fmt.Printf("Error: %s", aerr.Error())
            }
        } else {
            fmt.Printf("Error: %s", err.Error())
        }
        return err
    } else {
        fmt.Println("ExecuteScheduledQuery is successful, below is the output:")
        fmt.Println(executeScheduledQueryOutput.GoString())
        return nil
    }
 }
```

Python

```
def execute_scheduled_query(self, scheduled_query_arn, invocation_time):
    print("\nExecuting Scheduled Query")
    try:

 self.query_client.execute_scheduled_query(ScheduledQueryArn=scheduled_query_arn,
 InvocationTime=invocation_time)
        print("Successfully started executing scheduled query")
    except self.query_client.exceptions.ResourceNotFoundException as err:
        print("Scheduled Query doesn't exist")
        raise err
    except Exception as err:
        print("Scheduled Query execution failed:", err)
        raise err
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function executeScheduledQuery(scheduledQueryArn, invocationTime) {
     console.log("Executing Scheduled Query");
     var params = {
         ScheduledQueryArn: scheduledQueryArn,
         InvocationTime: invocationTime
     }
     try {
         await queryClient.executeScheduledQuery(params).promise();
     } catch (err) {
         console.log("Execute Scheduled Query failed: ", err);
         throw err;
     }
 }
```

.NET

```
private async Task ExecuteScheduledQuery(string scheduledQueryArn, DateTime
 invocationTime)
 {
     try
     {
         Console.WriteLine("Running Scheduled Query");
         await _amazonTimestreamQuery.ExecuteScheduledQueryAsync(new
 ExecuteScheduledQueryRequest()
         {
             ScheduledQueryArn = scheduledQueryArn,
             InvocationTime = invocationTime
         });
         Console.WriteLine("Successfully started manual run of scheduled query");
     }
     catch (ResourceNotFoundException e)
     {
         Console.WriteLine($"Scheduled Query doesn't exist: {e}");
         throw;
     }
     catch (Exception e)
     {
         Console.WriteLine($"Execute Scheduled Query failed: {e}");
```

```
        throw;
    }
}
```

Update scheduled query

You can use the following code snippets to update a scheduled query.

Java

```
public void updateScheduledQueries(String scheduledQueryArn) {
    System.out.println("Updating Scheduled Query");
    try {
        queryClient.updateScheduledQuery(new UpdateScheduledQueryRequest()
                .withScheduledQueryArn(scheduledQueryArn)
                .withState(ScheduledQueryState.DISABLED));
        System.out.println("Successfully update scheduled query state");
    }
    catch (ResourceNotFoundException e) {
        System.out.println("Scheduled Query doesn't exist");
        throw e;
    }
    catch (Exception e) {
        System.out.println("Execution Scheduled Query failed: " + e);
        throw e;
    }
}
```

Java v2

```
public void updateScheduledQuery(String scheduledQueryArn, ScheduledQueryState
 state) {
    System.out.println("Updating Scheduled Query");
    try {
        queryClient.updateScheduledQuery(UpdateScheduledQueryRequest.builder()
                .scheduledQueryArn(scheduledQueryArn)
                .state(state)
                .build());
        System.out.println("Successfully update scheduled query state");
    }
    catch (ResourceNotFoundException e) {
        System.out.println("Scheduled Query doesn't exist");
```

```
        throw e;
    }
    catch (Exception e) {
        System.out.println("Execution Scheduled Query failed: " + e);
        throw e;
    }
}
```

Go

```
func (timestreamBuilder TimestreamBuilder) UpdateScheduledQuery(scheduledQueryArn
 string) error {

     updateScheduledQueryInput := &timestreamquery.UpdateScheduledQueryInput{
         ScheduledQueryArn: aws.String(scheduledQueryArn),
         State:             aws.String(timestreamquery.ScheduledQueryStateDisabled),
     }
     _, err :=
 timestreamBuilder.QuerySvc.UpdateScheduledQuery(updateScheduledQueryInput)

     if err != nil {
         if aerr, ok := err.(awserr.Error); ok {
             switch aerr.Code() {
             case timestreamquery.ErrCodeResourceNotFoundException:
                 fmt.Println(timestreamquery.ErrCodeResourceNotFoundException,
 aerr.Error())
             default:
                 fmt.Printf("Error: %s", aerr.Error())
             }
         } else {
             fmt.Printf("Error: %s", err.Error())
         }
         return err
     } else {
         fmt.Println("UpdateScheduledQuery is successful")
         return nil
     }
 }
```

Python

```
def update_scheduled_query(self, scheduled_query_arn, state):
    print("\nUpdating Scheduled Query")
```

```
    try:

 self.query_client.update_scheduled_query(ScheduledQueryArn=scheduled_query_arn,
                                                  State=state)
        print("Successfully update scheduled query state to", state)
    except self.query_client.exceptions.ResourceNotFoundException as err:
        print("Scheduled Query doesn't exist")
        raise err
    except Exception as err:
        print("Scheduled Query deletion failed:", err)
        raise err
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function updateScheduledQueries(scheduledQueryArn) {
     console.log("Updating Scheduled Query");
     var params = {
         ScheduledQueryArn: scheduledQueryArn,
         State: "DISABLED"
     }
     try {
         await queryClient.updateScheduledQuery(params).promise();
         console.log("Successfully update scheduled query state");
     } catch (err) {
         console.log("Update Scheduled Query failed: ", err);
         throw err;
     }
 }
```

.NET

```
private async Task UpdateScheduledQuery(string scheduledQueryArn,
 ScheduledQueryState state)
 {
     try
     {
         Console.WriteLine("Updating Scheduled Query");
         await _amazonTimestreamQuery.UpdateScheduledQueryAsync(new
 UpdateScheduledQueryRequest()
         {
```

```
            ScheduledQueryArn = scheduledQueryArn,
            State = state
        });
        Console.WriteLine("Successfully update scheduled query state");
    }
    catch (ResourceNotFoundException e)
    {
        Console.WriteLine($"Scheduled Query doesn't exist: {e}");
        throw;
    }
    catch (Exception e)
    {
        Console.WriteLine($"Update Scheduled Query failed: {e}");
        throw;
    }
}
```

Delete scheduled query

You can use the following code snippets to delete a scheduled query.

Java

```
public void deleteScheduledQuery(String scheduledQueryArn) {
    System.out.println("Deleting Scheduled Query");

    try {
        queryClient.deleteScheduledQuery(new
 DeleteScheduledQueryRequest().withScheduledQueryArn(scheduledQueryArn));
        System.out.println("Successfully deleted scheduled query");
    }
    catch (Exception e) {
        System.out.println("Scheduled Query deletion failed: " + e);
    }
}
```

Java v2

```
public void deleteScheduledQuery(String scheduledQueryArn) {
    System.out.println("Deleting Scheduled Query");

    try {
```

```
        queryClient.deleteScheduledQuery(DeleteScheduledQueryRequest.builder()
                .scheduledQueryArn(scheduledQueryArn).build());
        System.out.println("Successfully deleted scheduled query");
    }
    catch (Exception e) {
        System.out.println("Scheduled Query deletion failed: " + e);
    }
}
```

Go

```
func (timestreamBuilder TimestreamBuilder) DeleteScheduledQuery(scheduledQueryArn
 string) error {

    deleteScheduledQueryInput := &timestreamquery.DeleteScheduledQueryInput{
        ScheduledQueryArn: aws.String(scheduledQueryArn),
    }
    _, err :=
 timestreamBuilder.QuerySvc.DeleteScheduledQuery(deleteScheduledQueryInput)

    if err != nil {
        fmt.Println("Error:")
        if aerr, ok := err.(awserr.Error); ok {
            switch aerr.Code() {
            case timestreamquery.ErrCodeResourceNotFoundException:
                fmt.Println(timestreamquery.ErrCodeResourceNotFoundException,
 aerr.Error())
            default:
                fmt.Printf("Error: %s", aerr.Error())
            }
        } else {
            fmt.Printf("Error: %s", err.Error())
        }
        return err
    } else {
        fmt.Println("DeleteScheduledQuery is successful")
        return nil
    }
 }
```

Python

```
def delete_scheduled_query(self, scheduled_query_arn):
```

```
    print("\nDeleting Scheduled Query")
    try:

 self.query_client.delete_scheduled_query(ScheduledQueryArn=scheduled_query_arn)
        print("Successfully deleted scheduled query :", scheduled_query_arn)
    except Exception as err:
        print("Scheduled Query deletion failed:", err)
        raise err
```

Node.js

The following snippet uses the AWS SDK for JavaScript V2 style. It is based on the sample application at Node.js sample Amazon Timestream for LiveAnalytics application on GitHub.

```
async function deleteScheduleQuery(scheduledQueryArn) {
     console.log("Deleting Scheduled Query");
     const params = {
         ScheduledQueryArn: scheduledQueryArn
     }
     try {
         await queryClient.deleteScheduledQuery(params).promise();
         console.log("Successfully deleted scheduled query");
     } catch (err) {
         console.log("Scheduled Query deletion failed: ", err);
     }
 }
```

.NET

```
private async Task DeleteScheduledQuery(string scheduledQueryArn)
 {
     try
     {
         Console.WriteLine("Deleting Scheduled Query");
         await _amazonTimestreamQuery.DeleteScheduledQueryAsync(new
 DeleteScheduledQueryRequest()
         {
             ScheduledQueryArn = scheduledQueryArn
         });
         Console.WriteLine($"Successfully deleted scheduled query :
 {scheduledQueryArn}");
     }
     catch (Exception e)
```

```
    {
        Console.WriteLine($"Scheduled Query deletion failed: {e}");
        throw;
    }
}
```

Using batch load in Timestream for LiveAnalytics

With *batch load* for Amazon Timestream for LiveAnalytics, you can ingest CSV files stored in Amazon S3 into Timestream in batches. With this new functionality, you can have your data in Timestream for LiveAnalytics without having to rely on other tools or write custom code. You can use batch load for backfilling data with flexible wait times, such as data that isn't immediately required for querying or analysis.

You can create batch load tasks by using the AWS Management Console, the AWS CLI, and the AWS SDKs. For more information, see Using batch load with the console, Using batch load with the AWS CLI, and Using batch load with the AWS SDKs.

In addition to batch load, you can write multiple records at the same time with the WriteRecords API operation. For guidance about which to use, see Choosing between the WriteRecords API operation and batch load.

Topics

- Batch load concepts in Timestream
- Batch load prerequisites
- Batch load best practices
- Preparing a batch load data file
- Data model mappings for batch load
- Using batch load with the console
- Using batch load with the AWS CLI
- Using batch load with the AWS SDKs
- Using batch load error reports

Batch load concepts in Timestream

Review the following concepts to better understand batch load functionality.

Batch load task – The task that defines your source data and destination in Amazon Timestream. You specify additional configuration such as the data model when you create the batch load task. You can create batch load tasks through the AWS Management Console, the AWS CLI, and the AWS SDKs.

Import destination – The destination database and table in Timestream. For information about creating databases and tables, see Create a database and Create a table.

Data source – The source CSV file that is stored in an S3 bucket. For information about preparing the data file, see Preparing a batch load data file. For information about S3 pricing, see Amazon S3 pricing.

Batch load error report – A report that stores information about the errors of a batch load task. You define the S3 location for batch load error reports as part of a batch load task. For information about information in the reports, see Using batch load error reports.

Data model mapping – A batch load mapping for time, dimensions, and measures that is from a data source in an S3 location to a target Timestream for LiveAnalytics table. For more information, see Data model mappings for batch load.

Batch load prerequisites

This is a list of prerequisites for using batch load. For best practices, see Batch load best practices.

- Batch load source data is stored in Amazon S3 in CSV format with headers.
- For each Amazon S3 source bucket, you must have the following permissions in an attached policy:

  ```
  "s3:GetObject",
  "s3:GetBucketAcl"
  "s3:ListBucket"
  ```

  Similarly, for each Amazon S3 output bucket where reports are written, you must have the following permissions in an attached policy:

  ```
  "s3:PutObject",
  "s3:GetBucketAcl"
  ```

  For example:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:GetObject",
                "s3:GetBucketAcl",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-source-bucket1\u201d",
                "arn:aws:s3:::amzn-s3-demo-source-bucket2\u201d"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "s3:PutObject",
                "s3:GetBucketAcl"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-destination-bucket\u201d"
            ],
            "Effect": "Allow"
        }
    ]
}
```

- Timestream for LiveAnalytics parses the CSV by mapping information that's provided in the data model to CSV headers. The data must have a column that represents the timestamp, at least one dimension column, and at least one measure column.
- The S3 buckets used with batch load must be in the same region and from the same account as the Timestream for LiveAnalytics table that is used in batch load.
- The `timestamp` column must be a long data type that represents the time since the Unix epoch. For example, the timestamp `2021-03-25T08:45:21Z` would be represented as `1616661921`. Timestream supports seconds, milliseconds, microseconds, and nanoseconds for the timestamp precision. When using the query language, you can convert between formats with functions such as `to_unixtime`. For more information, see Date / time functions.

- Timestream supports the string data type for dimension values. It supports long, double, string, and boolean data types for measure columns.

For batch load limits and quotas, see Batch load.

Batch load best practices

Batch load works best (high throughput) when adhering to the following conditions and recommendations:

1. CSV files submitted for ingestion are small, specifically with a file size of 100 MB–1 GB, to improve parallelism and speed of ingestion.
2. Avoid simultaneously ingesting data into the same table (e.g. using the WriteRecords API operation, or a scheduled query) when the batch load is in progress. This might lead to throttles, and the batch load task will fail.
3. Do not add, modify, or remove files from the S3 bucket used in batch load while the batch load task is running.
4. Do not delete or revoke permissions from tables or source, or report S3 buckets that have scheduled or in-progress batch load tasks.
5. When ingesting data with a high cardinality set of dimension values, follow guidance at Recommendations for partitioning multi-measure records.
6. Make sure you test the data for correctness by submitting a small file. You will be charged for any data submitted to batch load regardless of correctness. For more information about pricing, see Amazon Timestream pricing.
7. Do not resume a batch load task unless `ActiveMagneticStorePartitions` are below 250. The job may be throttled and fail. Submiting multiple jobs at the same time for the same database should reduce the number.

The following are console best practices:

1. Use the builder only for simpler data modeling that uses only one measure name for multi-measure records.
2. For more complex data modeling, use JSON. For example, use JSON when you use multiple measure names when using multi-measure records.

For additional Timestream for LiveAnalytics best practices, see Best practices.

Preparing a batch load data file

A source data file has delimiter-separated values. The more specific term, comma-separated values (CSV) is used generically. Valid column separators include commas and pipes. Records are separated by new lines. Files must be stored in Amazon S3. When you create a new batch load task, the location of the source data is specified by an ARN for the file. A file contains headers. One column represents the timestamp. At least one other column represents a measure.

The S3 buckets used with batch load must be in the same Region as the Timestream for LiveAnalytics table that is used in batch load. Don't add or remove files from the S3 bucket used in batch load after the batch load task has been submitted. For information about working with S3 buckets, see Getting started with Amazon S3.

> **Note**
>
> CSV files that are generated by some applications such as Excel might contain a byte order mark (BOM) that conflicts with the expected encoding. Timestream for LiveAnalytics batch load tasks that reference a CSV file with a BOM throw an error when they're processed programmatically. To avoid this, you can remove the BOM, which is an invisible character. For example, you can save the file from an application such as Notepad++ that lets you specify a new encoding. You can also use a programmatic option that reads the first line, removes the character from the line, and writes the new value over the first line in the file. When saving from Excel, there are multiple CSV options. Saving with a different CSV option might prevent the described issue. But you should check the result because a change in encoding can affect some characters.

CSV format parameters

You use escape characters when you're representing a value that is otherwise reserved by the format parameters. For example, if the quote character is a double quote, to represent a double quote in the data, place the escape character before the double quote.

For information about when to specify these when creating a batch load task, see Create a batch load task.

Parameter	Options
Column separator	(Comma (',') | Pipe ('|') | Semicolon (';') | Tab ('/t') | Blank space (' '))
Escape character	none
Quote character	Console: (Double quote (") | Single quote ('))
Null value	Blank space (' ')
Trim white space	Console: (No | Yes)

Data model mappings for batch load

The following discusses the schema for data model mappings and gives and example.

Data model mappings schema

The `CreateBatchLoadTask` request syntax and a `BatchLoadTaskDescription` object returned by a call to `DescribeBatchLoadTask` include a `DataModelConfiguration` object that includes the `DataModel` for batch loading. The `DataModel` defines mappings from source data that's stored in CSV format in an S3 location to a target Timestream for LiveAnalytics database and table.

The `TimeColumn` field indicates the source data's location for the value to be mapped to the destination table's `time` column in Timestream for LiveAnalytics. The `TimeUnit` specifies the unit for the `TimeColumn`, and can be one of `MILLISECONDS`, `SECONDS`, `MICROSECONDS`, or `NANOSECONDS`. There are also mappings for dimensions and measures. Dimension mappings are composed of source columns and target fields.

For more information, see DimensionMapping. The mappings for measures have two options, `MixedMeasureMappings` and `MultiMeasureMappings`.

To summarize, a `DataModel` contains mappings from a data source in an S3 location to a target Timestream for LiveAnalytics table for the following.

- Time
- Dimensions

- Measures

If possible, we recommend that you map measure data to multi-measure records in Timestream for LiveAnalytics. For information about the benefits of multi-measure records, see Multi-measure records.

If multiple measures in the source data are stored in one row, you can map those multiple measures to multi-measure records in Timestream for LiveAnalytics using `MultiMeasureMappings`. If there are values that must map to a single-measure record, you can use `MixedMeasureMappings`.

`MixedMeasureMappings` and `MultiMeasureMappings` both include `MultiMeasureAttributeMappings`. Multi-measure records are supported regardless of whether single-measure records are needed.

If only multi-measure target records are needed in Timestream for LiveAnalytics, you can define measure mappings in the following structure.

```
CreateBatchLoadTask
    MeasureNameColumn
    MultiMeasureMappings
        TargetMultiMeasureName
        MultiMeasureAttributeMappings array
```

> **Note**
>
> We recommend using `MultiMeasureMappings` whenever possible.

If single-measure target records are needed in Timestream for LiveAnalytics, you can define measure mappings in the following structure.

```
CreateBatchLoadTask
    MeasureNameColumn
    MixedMeasureMappings array
        MixedMeasureMapping
            MeasureName
            MeasureValueType
            SourceColumn
            TargetMeasureName
```

```
            MultiMeasureAttributeMappings array
```

When you use `MultiMeasureMappings`, the `MultiMeasureAttributeMappings` array is always required. When you use the `MixedMeasureMappings` array, if the `MeasureValueType` is `MULTI` for a given `MixedMeasureMapping`, `MultiMeasureAttributeMappings` is required for that `MixedMeasureMapping`. Otherwise, `MeasureValueType` indicates the measure type for the single-measure record.

Either way, there is an array of `MultiMeasureAttributeMapping` available. You define the mappings to multi-measure records in each `MultiMeasureAttributeMapping` as follows:

`SourceColumn`

The column in the source data that is located in Amazon S3.

`TargetMultiMeasureAttributeName`

The name of the target multi-measure name in the destination table. This input is required when `MeasureNameColumn` is not provided. If `MeasureNameColumn` is provided, the value from that column is used as the multi-measure name.

`MeasureValueType`

One of `DOUBLE`, `BIGINT` `BOOLEAN`, `VARCHAR`, or `TIMESTAMP`.

Data model mappings with `MultiMeasureMappings` example

This example demonstrates mapping to multi-measure records, the preferred approach, which store each measure value in a dedicated column. You can download a sample CSV at sample CSV. The sample has the following headings to map to a target column in a Timestream for LiveAnalytics table.

- `time`
- `measure_name`
- `region`
- `location`
- `hostname`
- `memory_utilization`
- `cpu_utilization`

Identify the `time` and `measure_name` columns in the CSV file. In this case these map directly to the Timestream for LiveAnalytics table columns of the same names.

- `time` maps to `time`
- `measure_name` maps to `measure_name` (or your chosen value)

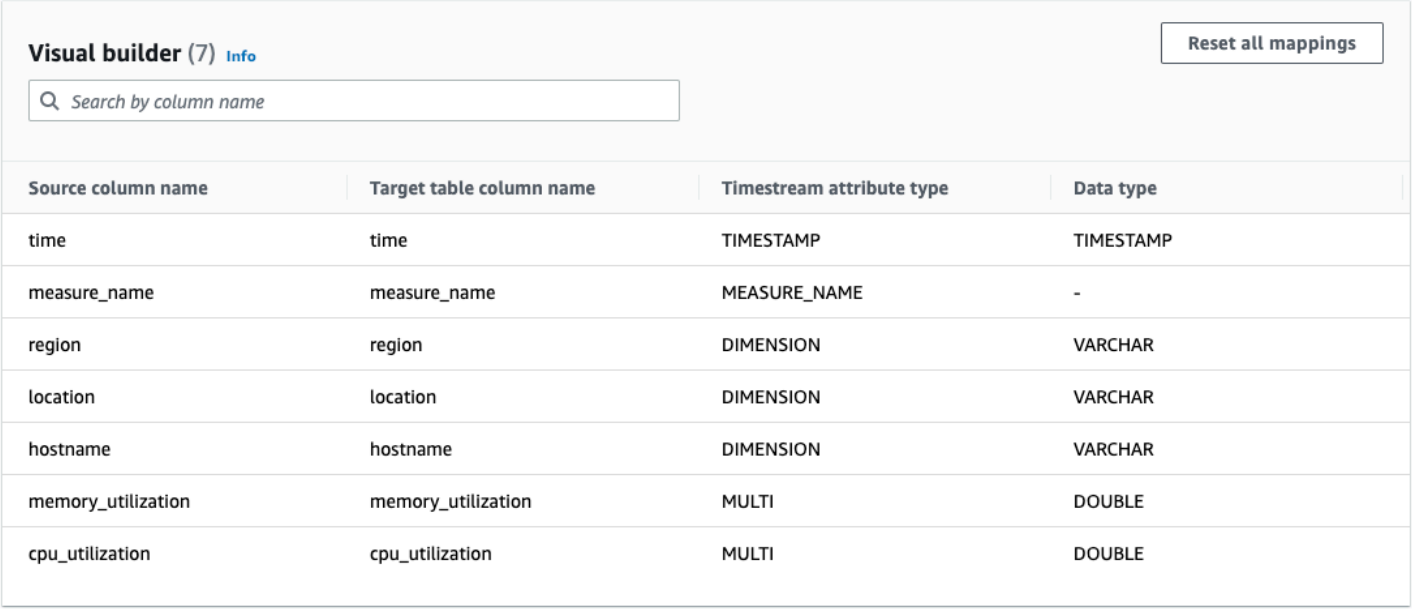
When using the API, you specify `time` in the `TimeColumn` field and a supported time unit value such as `MILLISECONDS` in the `TimeUnit` field. These correspond to **Source columnn name** and **Timestamp time input** in the console. You can group or partition records using `measure_name` which is defined with the `MeasureNameColumn` key.

In the sample, `region`, `location`, and `hostname` are dimensions. Dimensions are mapped in an array of `DimensionMapping` objects.

For measures, the value `TargetMultiMeasureAttributeName` will become a column in the Timestream for LiveAnalytics table. You can keep the same name such as in this example. Or you can specify a new one. `MeasureValueType` is one of `DOUBLE`, `BIGINT`, `BOOLEAN`, `VARCHAR`, or `TIMESTAMP`.

```
{
  "TimeColumn": "time",
  "TimeUnit": "MILLISECONDS",
  "DimensionMappings": [
    {
      "SourceColumn": "region",
      "DestinationColumn": "region"
    },
    {
      "SourceColumn": "location",
      "DestinationColumn": "location"
    },
    {
      "SourceColumn": "hostname",
      "DestinationColumn": "hostname"
    }
  ],
  "MeasureNameColumn": "measure_name",
  "MultiMeasureMappings": {
    "MultiMeasureAttributeMappings": [
      {
        "SourceColumn": "memory_utilization",
```

```
        "TargetMultiMeasureAttributeName": "memory_utilization",
        "MeasureValueType": "DOUBLE"
      },
      {
        "SourceColumn": "cpu_utilization",
        "TargetMultiMeasureAttributeName": "cpu_utilization",
        "MeasureValueType": "DOUBLE"
      }
    ]
  }
}
```



Source column name	Target table column name	Timestream attribute type	Data type
time	time	TIMESTAMP	TIMESTAMP
measure_name	measure_name	MEASURE_NAME	-
region	region	DIMENSION	VARCHAR
location	location	DIMENSION	VARCHAR
hostname	hostname	DIMENSION	VARCHAR
memory_utilization	memory_utilization	MULTI	DOUBLE
cpu_utilization	cpu_utilization	MULTI	DOUBLE

Data model mappings with `MixedMeasureMappings` example

We recommend that you only use this approach when you need to map to single-measure records in Timestream for LiveAnalytics.

Using batch load with the console

Following are steps for using batch load with the AWS Management Console. You can download a sample CSV at sample CSV.

Topics

- Access batch load
- Create a batch load task

- Resume a batch load task
- Using the visual builder

Access batch load

Follow these steps to access batch load using the AWS Management Console.

1. Open the Amazon Timestream console.
2. In the navigation pane, choose **Management Tools**, and then choose **Batch load tasks**.
3. From here, you can view the list of batch load tasks and drill into a given task for more details. You can also create and resume tasks.

Create a batch load task

Follow these steps to create a batch load task using the AWS Management Console.

1. Open the Amazon Timestream console.
2. In the navigation pane, choose **Management Tools**, and then choose **Batch load tasks**.
3. Choose **Create batch load task**.
4. In **Import destination**, choose the following.

   - **Target database** – Select the name of the database created in Create a database.
   - **Target table** – Select the name of the table created in Create a table.

   If necessary, you can add a table from this panel with the **Create new table** button.
5. From **Data source S3 location** in **Data source**, select the S3 bucket where the source data is stored. Use the **Browse S3** button to view S3 resources the active AWS account has access to, or enter the S3 location URL. The data source must be located in the same region.
6. In **File format settings** (expandable section), you can use the default settings to parse input data. You can also choose **Advanced settings**. From there you can choose **CSV format parameters**, and select parameters to parse input data. For information about these parameters, see CSV format parameters.
7. From **Configure data model mapping**, configure the data model. For additional data model guidance, see Data model mappings for batch load

- From **Data model mapping**, choose **Mapping configuration input**, and choose one of the following.
  - **Visual builder** – To map data visually, choose **TargetMultiMeasureName** or **MeasureNameColumn**. Then from **Visual builder**, map the columns.

    Visual builder automatically detects and loads the source column headers from the data source file when a single CSV file is selected as the data source. Choose the attribute and data type to create your mapping.

    For information about using the visual builder, see Using the visual builder.
  - **JSON editor** – A freeform JSON editor for configuring your data model. Choose this option if you're familiar with Timestream for LiveAnalytics and want to build advanced data model mappings.
  - **JSON file from S3** – Select a JSON model file you have stored in S3. Choose this option if you've already configured a data model and want to reuse it for additional batch loads.

8. From **Error logs S3 location** in **Error log report**, select the S3 location that will be used to report errors. For information about how to use this report, see Using batch load error reports.
9. For **Encryption key type**, choose one of the following.
   - **Amazon S3-managed key (SSE-S3)** – An encryption key that Amazon S3 creates, manages, and uses for you.
   - **AWS KMS key (SSE-KMS)** – An encryption key protected by AWS Key Management Service (AWS KMS).
10. Choose **Next**.
11. On the **Review and create page**, review the settings and edit as necessary.

    > **Note**
    >
    > You can't change batch load task settings after the task has been created. Task completion times will vary based on the amount of data being imported.

12. Choose **Create batch load task**.

Resume a batch load task

When you select a batch load task with a status of "Progress stopped" which is still resumable, you are prompted to resume the task. There is also a banner with a **Resume task** button when you view the details for those tasks. Resumable tasks have a "resume by" date. After that date expires, tasks cannot be resumed.

Using the visual builder

You can use the visual builder to map source data columns one or more CSV file(s) stored in an S3 bucket to destination columns in a Timestream for LiveAnalytics table.

> **Note**
>
> Your role will need the `SelectObjectContent` permission for the file. Without this, you will need to add and delete columns manually.

Auto load source columns mode

Timestream for LiveAnalytics can automatically scan the source CSV file for column names if you specify one bucket only. When there are no existing mappings, you can choose **Import source columns**.

1. With the **Visual builder** option selected from the **Mapping configuration input settings**, set the Timestamp time input. `Milliseconds` is the default setting.
2. Click the **Load source columns** button to import the column headers found in the source data file. The table will be populated with the source column header names from the data source file.
3. Choose the **Target table column name**, **Timestream attribute type**, and **Data type** for each source column.

   For details about these columns and possible values, see Mapping fields.
4. Use the drag-to-fill feature to set the value for multiple columns at once.

Manually add source columns

If you're using a bucket or CSV prefix and not a single CSV, you can add and delete column mappings from the visual editor with the **Add column mapping** and **Delete column mapping** buttons. There is also a button to reset mappings.

Mapping fields

- **Source column name** – The name of a column in the source file that represents a measure to import. Timestream for LiveAnalytics can populate this value automatically when you use **Import source columns**.
- **Target table column name** – Optional input that indicates the column name for the measure in the target table.
- **Timestream attribute type** – The attribute type of the data in the specified source column such as `DIMENSION`.
  - **TIMESTAMP** – Specifies when a measure was collected.
  - **MULTI** – Multiple measures are represented.
  - **DIMENSION** – Time series metadata.
  - **MEASURE_NAME** – For single-measure records, this is the measure name.
- **Data type** – The type of Timestream column, such as `BOOLEAN`.
  - **BIGINT** – A 64-bit integer.
  - **BOOLEAN** – The two truth values of logic—true and false.
  - **DOUBLE** – 64-bit variable-precision number.
  - **TIMESTAMP** – An instance in time that uses nanosecond precision time in UTC, and tracks the time since the Unix epoch.

Using batch load with the AWS CLI

Setup

To start using batch load, go through the following steps.

1. Install the AWS CLI using the instructions at Accessing Amazon Timestream for LiveAnalytics using the AWS CLI.
2. Run the following command to verify that the Timestream CLI commands have been updated. Verify that create-batch-load-task is in the list.

```
aws timestream-write help
```

3. Prepare a data source using the instructions at Preparing a batch load data file.
4. Create a database and table using the instructions at Accessing Amazon Timestream for LiveAnalytics using the AWS CLI.
5. Create an S3 bucket for report output. The bucket must be in the same Region. For more information about buckets, see Creating, configuring, and working with Amazon S3 buckets.
6. Create a batch load task. For steps, see Create a batch load task.
7. Confirm the status of the task. For steps, see Describe batch load task.

Create a batch load task

You can create a batch load task with the `create-batch-load-task` command. When you create a batch load task using the CLI, you can use a JSON parameter, `cli-input-json`, which lets you aggregate the parameters into a single JSON fragment. You can also break those details apart using several other parameters including `data-model-configuration`, `data-source-configuration`, `report-configuration`, `target-database-name`, and `target-table-name`.

For an example, see Create batch load task example

Describe batch load task

You can retrieve a batch load task description as follows.

```
aws timestream-write describe-batch-load-task --task-id <value>
```

Following is an example response.

```
{
    "BatchLoadTaskDescription": {
        "TaskId": "<TaskId>",
        "DataSourceConfiguration": {
            "DataSourceS3Configuration": {
                "BucketName": "test-batch-load-west-2",
                "ObjectKeyPrefix": "sample.csv"
            },
            "CsvConfiguration": {},
            "DataFormat": "CSV"
```

```
        },
        "ProgressReport": {
            "RecordsProcessed": 2,
            "RecordsIngested": 0,
            "FileParseFailures": 0,
            "RecordIngestionFailures": 2,
            "FileFailures": 0,
            "BytesIngested": 119
        },
        "ReportConfiguration": {
            "ReportS3Configuration": {
                "BucketName": "test-batch-load-west-2",
                "ObjectKeyPrefix": "<ObjectKeyPrefix>",
                "EncryptionOption": "SSE_S3"
            }
        },
        "DataModelConfiguration": {
            "DataModel": {
                "TimeColumn": "timestamp",
                "TimeUnit": "SECONDS",
                "DimensionMappings": [
                    {
                        "SourceColumn": "vehicle",
                        "DestinationColumn": "vehicle"
                    },
                    {
                        "SourceColumn": "registration",
                        "DestinationColumn": "license"
                    }
                ],
                "MultiMeasureMappings": {
                    "TargetMultiMeasureName": "test",
                    "MultiMeasureAttributeMappings": [
                        {
                            "SourceColumn": "wgt",
                            "TargetMultiMeasureAttributeName": "weight",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
                            "SourceColumn": "spd",
                            "TargetMultiMeasureAttributeName": "speed",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
```

```
                            "SourceColumn": "fuel",
                            "TargetMultiMeasureAttributeName": "fuel",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
                            "SourceColumn": "miles",
                            "TargetMultiMeasureAttributeName": "miles",
                            "MeasureValueType": "DOUBLE"
                        }
                    ]
                }
            }
        },
        "TargetDatabaseName": "BatchLoadExampleDatabase",
        "TargetTableName": "BatchLoadExampleTable",
        "TaskStatus": "FAILED",
        "RecordVersion": 1,
        "CreationTime": 1677167593.266,
        "LastUpdatedTime": 1677167602.38
    }
}
```

List batch load tasks

You can list batch load tasks as follows.

```
aws timestream-write list-batch-load-tasks
```

An output appears as follows.

```
{
    "BatchLoadTasks": [
        {
            "TaskId": "<TaskId>",
            "TaskStatus": "FAILED",
            "DatabaseName": "BatchLoadExampleDatabase",
            "TableName": "BatchLoadExampleTable",
            "CreationTime": 1677167593.266,
            "LastUpdatedTime": 1677167602.38
        }
    ]
}
```

Resume batch load task

You can resume a batch load task as follows.

```
aws timestream-write resume-batch-load-task --task-id <value>
```

A response can indicate success or contain error information.

Create batch load task example

Example

1. Create a Timestream for LiveAnalytics database named `BatchLoad` and a table named `BatchLoadTest`. Verify and, if necessary, adjust the values for `MemoryStoreRetentionPeriodInHours` and `MagneticStoreRetentionPeriodInDays`.

   ```
   aws timestream-write create-database --database-name BatchLoad \

   aws timestream-write create-table --database-name BatchLoad \
   --table-name BatchLoadTest \
   --retention-properties "{\"MemoryStoreRetentionPeriodInHours\": 12,
    \"MagneticStoreRetentionPeriodInDays\": 100}"
   ```

2. Using the console, create an S3 bucket and copy the `sample.csv` file to that location. You can download a sample CSV at sample CSV.
3. Using the console create an S3 bucket for Timestream for LiveAnalytics to write a report if the batch load task completes with errors.
4. Create a batch load task. Make sure to replace *$INPUT_BUCKET* and *$REPORT_BUCKET* with the buckets that you created in the preceding steps.

   ```
   aws timestream-write create-batch-load-task \
   --data-model-configuration "{\
               \"DataModel\": {\
                 \"TimeColumn\": \"timestamp\",\
                 \"TimeUnit\": \"SECONDS\",\
                 \"DimensionMappings\": [\
                   {\
                     \"SourceColumn\": \"vehicle\"\
                   },\
                   {\
                     \"SourceColumn\": \"registration\",\
   ```

```
                  \"DestinationColumn\": \"license\"\
                }\
              ],
              \"MultiMeasureMappings\": {\
                \"TargetMultiMeasureName\": \"mva_measure_name\",\
                \"MultiMeasureAttributeMappings\": [\
                  {\
                    \"SourceColumn\": \"wgt\",\
                    \"TargetMultiMeasureAttributeName\": \"weight\",\
                    \"MeasureValueType\": \"DOUBLE\"\
                  },\
                  {\
                    \"SourceColumn\": \"spd\",\
                    \"TargetMultiMeasureAttributeName\": \"speed\",\
                    \"MeasureValueType\": \"DOUBLE\"\
                  },\
                  {\
                    \"SourceColumn\": \"fuel_consumption\",\
                    \"TargetMultiMeasureAttributeName\": \"fuel\",\
                    \"MeasureValueType\": \"DOUBLE\"\
                  },\
                  {\
                    \"SourceColumn\": \"miles\",\
                    \"MeasureValueType\": \"BIGINT\"\
                  }\
                ]\
              }\
            }\
          }" \
--data-source-configuration "{
            \"DataSourceS3Configuration\": {\
              \"BucketName\": \"$INPUT_BUCKET\",\
              \"ObjectKeyPrefix\": \"$INPUT_OBJECT_KEY_PREFIX\"
            },\
            \"DataFormat\": \"CSV\"\
          }" \
--report-configuration "{\
            \"ReportS3Configuration\": {\
              \"BucketName\": \"$REPORT_BUCKET\",\
              \"EncryptionOption\": \"SSE_S3\"\
            }\
          }" \
--target-database-name BatchLoad \
```

```
--target-table-name BatchLoadTest
```

The preceding command returns the following output.

```
{
    "TaskId": "TaskId "
}
```

5. Check on the progress of the task. Make sure you replace *$TASK_ID* with the task id that was returned in the preceding step.

```
aws timestream-write describe-batch-load-task --task-id $TASK_ID
```

Example output

```
{
    "BatchLoadTaskDescription": {
        "ProgressReport": {
            "BytesIngested": 1024,
            "RecordsIngested": 2,
            "FileFailures": 0,
            "RecordIngestionFailures": 0,
            "RecordsProcessed": 2,
            "FileParseFailures": 0
        },
        "DataModelConfiguration": {
            "DataModel": {
                "DimensionMappings": [
                    {
                        "SourceColumn": "vehicle",
                        "DestinationColumn": "vehicle"
                    },
                    {
                        "SourceColumn": "registration",
                        "DestinationColumn": "license"
                    }
                ],
                "TimeUnit": "SECONDS",
                "TimeColumn": "timestamp",
                "MultiMeasureMappings": {
                    "MultiMeasureAttributeMappings": [
```

```
                        {
                            "TargetMultiMeasureAttributeName": "weight",
                            "SourceColumn": "wgt",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
                            "TargetMultiMeasureAttributeName": "speed",
                            "SourceColumn": "spd",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
                            "TargetMultiMeasureAttributeName": "fuel",
                            "SourceColumn": "fuel_consumption",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
                            "TargetMultiMeasureAttributeName": "miles",
                            "SourceColumn": "miles",
                            "MeasureValueType": "DOUBLE"
                        }
                    ],
                    "TargetMultiMeasureName": "mva_measure_name"
                }
            }
        },
        "TargetDatabaseName": "BatchLoad",
        "CreationTime": 1672960381.735,
        "TaskStatus": "SUCCEEDED",
        "RecordVersion": 1,
        "TaskId": "TaskId ",
        "TargetTableName": "BatchLoadTest",
        "ReportConfiguration": {
            "ReportS3Configuration": {
                "EncryptionOption": "SSE_S3",
                "ObjectKeyPrefix": "ObjectKeyPrefix ",
                "BucketName": "amzn-s3-demo-bucket"
            }
        },
        "DataSourceConfiguration": {
            "DataSourceS3Configuration": {
                "ObjectKeyPrefix": "sample.csv",
                "BucketName": "amzn-s3-demo-source-bucket"
            },
            "DataFormat": "CSV",
```

```
            "CsvConfiguration": {}
        },
        "LastUpdatedTime": 1672960387.334
    }
}
```

Using batch load with the AWS SDKs

For examples of how to create, describe, and list batch load tasks with the AWS SDKs, see Create batch load task, Describe batch load task, List batch load tasks, and Resume batch load task.

Using batch load error reports

Batch load tasks have one of the following status values:

- `CREATED` (**Created**) – Task is created.
- `IN_PROGRESS` (**In progress**) – Task is in progress.
- `FAILED` (**Failed**) – Task has completed. But one or more errors was detected.
- `SUCCEEDED` (**Completed**) – Task has completed with no errors.
- `PROGRESS_STOPPED` (**Progress stopped**) – Task has stopped but not completed. You can attempt to resume the task.
- `PENDING_RESUME` (**Pending resume**) – The task is pending to resume.

When there are errors, an error log report is created in the S3 bucket defined for that. Errors are categorized as taskErrors or fileErrors in separate arrays. Following is an example error report.

```
{
    "taskId": "9367BE28418C5EF902676482220B631C",
    "taskErrors": [],
    "fileErrors": [
        {
            "fileName": "example.csv",
            "errors": [
                {
                    "reason": "The record timestamp is outside the time range of the
 data ingestion window.",
                    "lineRanges": [
                        [
                            2,
```

```
                                3
                            ]
                        ]
                    }
                ]
            }
        ]
}
```

Using scheduled queries in Timestream for LiveAnalytics

The scheduled query feature in Amazon Timestream for LiveAnalytics is a fully managed, serverless, and scalable solution for calculating and storing aggregates, rollups, and other forms of preprocessed data typically used for operational dashboards, business reports, ad-hoc analytics, and other applications. Scheduled queries make real-time analytics more performant and cost-effective, so you can derive additional insights from your data, and can continue to make better business decisions.

With scheduled queries, you define the real-time analytics queries that compute aggregates, rollups, and other operations on the data—and Amazon Timestream for LiveAnalytics periodically and automatically runs these queries and reliably writes the query results into a separate table. The data is typically calculated and updated into these tables within a few minutes.

You can then point your dashboards and reports to query the tables that contain aggregated data instead of querying the considerably larger source tables. This leads to performance and cost gains that can exceed orders of magnitude. This is because the tables with aggregated data contain much less data than the source tables, so they offer faster queries and cheaper data storage.

Additionally, tables with scheduled queries offer all of the existing functionality of a Timestream for LiveAnalytics table. For example, you can query the tables using SQL. You can visualize the data stored in the tables using Grafana. You can also ingest data into the table using Amazon Kinesis, Amazon MSK, AWS IoT Core, and Telegraf. You can configure data retention policies on these tables for automatic data lifecycle management.

Because the data retention of the tables that contain aggregated data is fully decoupled from that of source tables, you can also choose to reduce the data retention of the source tables and keep the aggregate data for a much longer duration, at a fraction of the data storage cost. Scheduled queries make real-time analytics faster, cheaper, and therefore more accessible to many more customers, so they can monitor their applications and drive better data-driven business decisions.

Topics

- Scheduled query benefits
- Scheduled query use cases
- Example: Using real-time analytics to detect fraudulent payments and make better business decisions
- Scheduled query concepts
- Schedule expressions for scheduled queries
- Data model mappings for scheduled queries
- Scheduled query notification messages
- Scheduled query error reports
- Scheduled query patterns and examples

Scheduled query benefits

The following are the benefits of scheduled queries:

- **Operational ease** – Scheduled queries are serverless and fully managed.
- **Performance and cost** – Because scheduled queries precompute the aggregates, rollups, or other real-time analytics operations for your data and store the results in a table, queries that access tables populated by scheduled queries contain less data than the source tables. Therefore, queries that are run on these tables are faster and cheaper. Tables populated by scheduled computations contain less data than their source tables, and therefore help reduce the storage cost. You can also retain this data for a longer duration in the memory store at a fraction of the cost of retaining the source data in the memory store.
- **Interoperability** – Tables populated by scheduled queries offer all of the existing functionality of Timestream for LiveAnalytics tables and can be used with all of the services and tools that work with Timestream for LiveAnalytics. See Working with Other Services for details.

Scheduled query use cases

You can use scheduled queries for business reports that summarize the end-user activity from your applications, so you can train machine learning models for personalization. You can also use scheduled queries for alarms that detect anomalies, network intrusions, or fraudulent activity, so you can take immediate remedial actions.

Additionally, you can use scheduled queries for more effective data governance. You can do this by granting source table access exclusively to the scheduled queries, and providing your developers access to only the tables populated by scheduled queries. This minimizes the impact of unintentional, long-running queries.

Example: Using real-time analytics to detect fraudulent payments and make better business decisions

Consider a payment system that processes transactions sent from multiple point-of-sale terminals distributed across major metropolitan cities in the United States. You want to use Amazon Timestream for LiveAnalytics to store and analyze the transaction data, so you can detect fraudulent transactions and run real-time analytics queries. These queries can help you answer business questions such as identifying the busiest and least used point-of-sale terminals per hour, the busiest hour of the day for each city, and the city with most transactions per hour.

The system process ~100K transactions per minute. Each transaction stored in Amazon Timestream for LiveAnalytics is 100 bytes. You've configured 10 queries that run every minute to detect various kinds of fraudulent payments. You've also created 25 queries that aggregate and slice/dice your data along various dimensions to help answer your business questions. Each of these queries processes the last hour's data.

You've created a dashboard to display the data generated by these queries. The dashboard contains 25 widgets, it is refreshed every hour, and it is typically accessed by 10 users at any given time. Finally, your memory store is configured with a 2-hour data retention period and the magnetic store is configured to have a 6-month data retention period.

In this case, you can use real-time analytics queries that recompute the data every time the dashboard is accessed and refreshed, or use derived tables for the dashboard. The query cost for dashboards based on real-time analytics queries will be $120.70 per month. In contrast, the cost of dashboarding queries powered by derived tables will be $12.27 per month (see Amazon Timestream for LiveAnalytics pricing). In this case, using derived tables reduces the query cost by ~10 times.

Scheduled query concepts

Query string - This is the query whose result you are pre-computing and storing in another Timestream for LiveAnalytics table. You can define a scheduled query using the full SQL surface area of Timestream for LiveAnalytics, which provides you the flexibility of writing queries with

common table expressions, nested queries, window functions, or any kind of aggregate and scalar functions that are supported by Timestream for LiveAnalytics query language.

Schedule expression - Allows you to specify when your scheduled query instances are run. You can specify the expressions using a cron expression (such as run at 8 AM UTC every day) or rate expression (such as run every 10 minutes).

Target configuration - Allows you to specify how you map the result of a scheduled query into the destination table where the results of this scheduled query will be stored.

Notification configuration -Timestream for LiveAnalytics automatically runs instances of a scheduled query based on your schedule expression. You receive a notification for every such query run on an SNS topic that you configure when you create a scheduled query. This notification specifies whether the instance was successfully run or encountered any errors. In addition, it provides information such as the bytes metered, data written to the target table, next invocation time, and so on.

The following is an example of this kind of notification message.

```
{
    "type":"AUTO_TRIGGER_SUCCESS",
    "arn":"arn:aws:timestream:us-east-1:123456789012:scheduled-query/
 PT1mPerMinutePerRegionMeasureCount-9376096f7309",
    "nextInvocationEpochSecond":1637302500,
    "scheduledQueryRunSummary":
    {
        "invocationEpochSecond":1637302440,
        "triggerTimeMillis":1637302445697,
        "runStatus":"AUTO_TRIGGER_SUCCESS",
        "executionStats":
        {
            "executionTimeInMillis":21669,
            "dataWrites":36864,
            "bytesMetered":13547036820,
            "recordsIngested":1200,
            "queryResultRows":1200
        }
    }
}
```

In this notification message, `bytesMetered` is the bytes that the query scanned on the source table, and dataWrites is the bytes written to the target table.

> **Note**
>
> If you are consuming these notifications programmatically, be aware that new fields could be added to the notification message in the future.

Error report location - Scheduled queries asynchronously run and store data in the target table. If an instance encounters any errors (for example, invalid data which could not be stored), the records that encountered errors are written to an error report in the error report location you specify at creation of a scheduled query. You specify the S3 bucket and prefix for the location. Timestream for LiveAnalytics appends the scheduled query name and invocation time to this prefix to help you identify the errors associated with a specific instance of a scheduled query.

Tagging - You can optionally specify tags that you can associate with a scheduled query. For more details, see Tagging Timestream for LiveAnalytics Resources.

Example

In the following example, you compute a simple aggregate using a scheduled query:

```
SELECT region, bin(time, 1m) as minute,
    SUM(CASE WHEN measure_name = 'metrics' THEN 20 ELSE 5 END) as numDataPoints
FROM raw_data.devops
WHERE time BETWEEN @scheduled_runtime - 10m AND @scheduled_runtime + 1m
GROUP BY bin(time, 1m), region
```

`@scheduled_runtime parameter` - In this example, you will notice the query accepting a special named parameter `@scheduled_runtime`. This is a special parameter (of type Timestamp) that the service sets when invoking a specific instance of a scheduled query so that you can deterministically control the time range for which a specific instance of a scheduled query analyzes the data in the source table. You can use `@scheduled_runtime` in your query in any location where a Timestamp type is expected.

Consider an example where you set a schedule expression: cron(0/5 * * * ? *) where the scheduled query will run at minute 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55 of every hour. For the instance that is triggered at 2021-12-01 00:05:00, the @scheduled_runtime parameter is initialized to this value, such that the instance at this time operates on data in the range 2021-11-30 23:55:00 to 2021-12-01 00:06:00.

Instances with overlapping time ranges - As you will see in this example, two subsequent instances of a scheduled query can overlap in their time ranges. This is something you can control based on your requirements, the time predicates you specify, and the schedule expression. In this case, this overlap allows these computations to update the aggregates based on any data whose arrival was slightly delayed, up to 10 minutes in this example. The query run triggered at 2021-12-01 00:00:00 will cover the time range 2021-11-30 23:50:00 to 2021-12-30 00:01:00 and the query run triggered at 2021-12-01 00:05:00 will cover the range 2021-11-30 23:55:00 to 2021-12-01 00:06:00.

To ensure correctness and to make sure that the aggregates stored in the target table match the aggregates computed from the source table, Timestream for LiveAnalytics ensures that the computation at 2021-12-01 00:05:00 will be performed only after the computation at 2021-12-01 00:00:00 has completed. The results of the latter computations can update any previously materialized aggregate if a newer value is generated. Internally, Timestream for LiveAnalytics uses record versions where records generated by latter instances of a scheduled query will be assigned a higher version number. Therefore, the aggregates computed by the invocation at 2021-12-01 00:05:00 can update the aggregates computed by the invocation at 2021-12-01 00:00:00, assuming newer data is available on the source table.

Automatic triggers vs. manual triggers - After a scheduled query is created, Timestream for LiveAnalytics will automatically run the instances based on the specified schedule. Such automated triggers are managed entirely by the service.

However, there might be scenarios where you might want to manually initiate some instances of a scheduled query. Examples include if a specific instance failed in a query run, if there was late-arriving data or updates in the source table after the automated schedule run, or if you want to update the target table for time ranges that are not covered by automated query runs (for example, for time ranges before creation of a scheduled query).

You can use the ExecuteScheduledQuery API to manually initiate a specific instance of a scheduled query by passing the InvocationTime parameter, which is a value used for the @scheduled_runtime parameter. The following are a few important considerations when using the ExecuteScheduledQuery API:

- If you are triggering multiple of these invocations, you need to make sure that these invocations do not generate results in overlapping time ranges. If you cannot ensure non-overlapping time ranges, then make sure that these query runs are initiated sequentially one after the other. If you concurrently initiate multiple query runs that overlap in their time ranges, then you can see trigger failures where you might see version conflicts in the error reports for these query runs.

- You can initiate the invocations with any timestamp value for @scheduled_runtime. So it is your responsibility to appropriately set the values so the appropriate time ranges are updated in the target table corresponding to the ranges where data was updated in the source table.
- The ExecuteScheduledQuery API operates asynchronously. Upon a successful call, the service sends a 200 response and proceeds to execute the query. However, if there are multiple scheduled query executions concurrently running, anticipate potential delays in executing manually triggered scheduled executions.

Schedule expressions for scheduled queries

You can create scheduled queries on an automated schedule by using Amazon Timestream for LiveAnalytics scheduled queries that use cron or rate expressions. All scheduled queries use the UTC time zone, and the minimum possible precision for schedules is 1 minute.

Two ways to specify the schedule expressions are *cron* and *rate*. Cron expressions offer more fine grained schedule control, while rate expressions are simpler to express but lack the fine-grained control.

For example, with a cron expression, you can define a scheduled query that gets triggered at a specified time on a certain day of each week or month, or a specified minute every hour only on Monday - Friday, and so on. In contrast, rate expressions initiate a scheduled query at a regular rate, such as once every minute, hour, or day, starting from the exact time when the scheduled query is created.

Cron expression

- *Syntax*

```
cron(fields)
```

Cron expressions have six required fields, which are separated by white space.

Field	Values	Wildcards
Minutes	0-59	, - * /
Hours	0-23	, - * /

Field	Values	Wildcards
Day-of-month	1-31	, - * ? / L W
Month	1-12 or JAN-DEC	, - * /
Day-of-week	1-7 or SUN-SAT	, - * ? L #
Year	1970-2199	, - * /

Wildcard characters

- The *,* (comma) wildcard includes additional values. In the Month field, JAN,FEB,MAR would include January, February, and March.
- The *-* (dash) wildcard specifies ranges. In the Day field, 1-15 would include days 1 through 15 of the specified month.
- The *** (asterisk) wildcard includes all values in the field. In the Hours field, *** would include every hour. You cannot use *** in both the Day-of-month and Day-of-week fields. If you use it in one, you must use *?* in the other.
- The */* (forward slash) wildcard specifies increments. In the Minutes field, you could enter 1/10 to specify every 10th minute, starting from the first minute of the hour (for example, the 11th, 21st, and 31st minute, and so on).
- The *?* (question mark) wildcard specifies one or another. In the Day-of-month field you could enter *7* and if you didn't care what day of the week the 7th was, you could enter *?* in the Day-of-week field.
- The *L* wildcard in the Day-of-month or Day-of-week fields specifies the last day of the month or week.
- The W wildcard in the Day-of-month field specifies a weekday. In the Day-of-month field, 3W specifies the weekday closest to the third day of the month.
- The *#* wildcard in the Day-of-week field specifies a certain instance of the specified day of the week within a month. For example, 3#2 would be the second Tuesday of the month: the 3 refers to Tuesday because it is the third day of each week, and the 2 refers to the second day of that type within the month.

> **Note**
>
> If you use a '#' character, you can define only one expression in the day-of-week field. For example, "3#1,6#3" is not valid because it is interpreted as two expressions.

Limitations

- You can't specify the Day-of-month and Day-of-week fields in the same cron expression. If you specify a value (or a *) in one of the fields, you must use a *?* (question mark) in the other.
- Cron expressions that lead to rates faster than 1 minute are not supported.

Examples

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
0	10	*	*	?	*	Run at 10:00 am (UTC) every day.
15	12	*	*	?	*	Run at 12:15 pm (UTC) every day.
0	18	?	*	MON-FRI	*	Run at 6:00 pm (UTC) every Monday through Friday.

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
0	8	1	*	?	*	Run at 8:00 am (UTC) every first day of the month.
0/15	*	*	*	?	*	Run every 15 minutes.
0/10	*	*	*	MON-FRI	*	Run every 10 minutes Monday through Friday.
0/5	8-17	?	*	MON-FRI	*	Run every 5 minutes Monday through Friday between 8:00 am and 5:55 pm (UTC).

Rate expressions

- A rate expression starts when you create the scheduled event rule, and then runs on its defined schedule. Rate expressions have two required fields. Fields are separated by white space.

  Syntax

```
rate(value unit)
```

- `value`: A positive number.
- `unit`: The unit of time. Different units are required for values of 1 (for example, minute) and values over 1 (for example, minutes). Valid values: minute | minutes | hour | hours | day | days

Data model mappings for scheduled queries

Timestream for LiveAnalytics supports flexible modeling of data in its tables and this same flexibility applies to results of scheduled queries that are materialized into another Timestream for LiveAnalytics table. With scheduled queries, you can query any table, whether it has data in multi-measure records or single-measure records and write the query results using either multi-measure or single-measure records.

You use the TargetConfiguration in the specification of a scheduled query to map the query results to the appropriate columns in the destination derived table. The following sections describe the different ways of specifying this TargetConfiguration to achieve different data models in the derived table. Specifically, you will see:

- How to write to multi-measure records when the query result does not have a measure name and you specify the target measure name in the TargetConfiguration.
- How you use measure name in the query result to write multi-measure records.
- How you can define a model to write multiple records with different multi-measure attributes.
- How you can define a model to write to single-measure records in the derived table.
- How you can query single-measure records and/or multi-measure records in a scheduled query and have the results materialized to either a single-measure record or a multi-measure record, which allows you to choose the flexibility of data models.

Example: Target measure name for multi-measure records

In this example, you will see that the query is reading data from a table with multi-measure data and is writing the results into another table using multi-measure records. The scheduled query result does not have a natural measure name column. Here, you specify the measure name in the derived table using the TargetMultiMeasureName property in the TargetConfiguration.TimestreamConfiguration.

```
{
    "Name" : "CustomMultiMeasureName",
    "QueryString" : "SELECT region, bin(time, 1h) as hour, AVG(memory_cached)
 as avg_mem_cached_1h, MIN(memory_free) as min_mem_free_1h, MAX(memory_used) as
 max_mem_used_1h, SUM(disk_io_writes) as sum_1h, AVG(disk_used) as avg_disk_used_1h,
 AVG(disk_free) as avg_disk_free_1h, MAX(cpu_user) as max_cpu_user_1h, MIN(cpu_idle) as
 min_cpu_idle_1h, MAX(cpu_system) as max_cpu_system_1h FROM raw_data.devops_multi WHERE
 time BETWEEN bin(@scheduled_runtime, 1h) - 14h AND bin(@scheduled_runtime, 1h) - 2h
 AND measure_name = 'metrics' GROUP BY region, bin(time, 1h)",
    "ScheduleConfiguration" : {
        "ScheduleExpression" : "cron(0 0/1 * * ? *)"
    },
    "NotificationConfiguration" : {
        "SnsConfiguration" : {
            "TopicArn" : "******"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******",
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName" : "derived",
            "TableName" : "dashboard_metrics_1h_agg_1",
            "TimeColumn" : "hour",
            "DimensionMappings" : [
                {
                    "Name": "region",
                    "DimensionValueType" : "VARCHAR"
                }
            ],
            "MultiMeasureMappings" : {
                "TargetMultiMeasureName": "dashboard-metrics",
                "MultiMeasureAttributeMappings" : [
                    {
                        "SourceColumn" : "avg_mem_cached_1h",
                        "MeasureValueType" : "DOUBLE",
                        "TargetMultiMeasureAttributeName" : "avgMemCached"
                    },
                    {
                        "SourceColumn" : "min_mem_free_1h",
                        "MeasureValueType" : "DOUBLE"
                    },
                    {
                        "SourceColumn" : "max_mem_used_1h",
```

```
                        "MeasureValueType" : "DOUBLE"
                    },
                    {
                        "SourceColumn" : "sum_1h",
                        "MeasureValueType" : "DOUBLE",
                        "TargetMultiMeasureAttributeName" : "totalDiskWrites"
                    },
                    {
                        "SourceColumn" : "avg_disk_used_1h",
                        "MeasureValueType" : "DOUBLE"
                    },
                    {
                        "SourceColumn" : "avg_disk_free_1h",
                        "MeasureValueType" : "DOUBLE"
                    },
                    {
                        "SourceColumn" : "max_cpu_user_1h",
                        "MeasureValueType" : "DOUBLE",
                        "TargetMultiMeasureAttributeName" : "CpuUserP100"
                    },
                    {
                        "SourceColumn" : "min_cpu_idle_1h",
                        "MeasureValueType" : "DOUBLE"
                    },
                    {
                        "SourceColumn" : "max_cpu_system_1h",
                        "MeasureValueType" : "DOUBLE",
                        "TargetMultiMeasureAttributeName" : "CpuSystemP100"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    }
}
```

The mapping in this example creates one multi-measure record with measure name dashboard-metrics and attribute names avgMemCached, min_mem_free_1h, max_mem_used_1h, totalDiskWrites, avg_disk_used_1h, avg_disk_free_1h, CpuUserP100, min_cpu_idle_1h, CpuSystemP100. Notice the optional use of TargetMultiMeasureAttributeName to rename the query output columns to a different attribute name used for result materialization.

The following is the schema for the destination table once this scheduled query is materialized. As you can see from the Timestream for LiveAnalytics attribute type in the following result, the results are materialized into a multi-measure record with a single-measure name `dashboard-metrics`, as shown in the measure schema.

Column	Type	Timestream for LiveAnalytics attribute type
region	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
CpuSystemP100	double	MULTI
avgMemCached	double	MULTI
min_cpu_idle_1h	double	MULTI
avg_disk_free_1h	double	MULTI
avg_disk_used_1h	double	MULTI
totalDiskWrites	double	MULTI
max_mem_used_1h	double	MULTI
min_mem_free_1h	double	MULTI
CpuUserP100	double	MULTI

The following are the corresponding measures obtained with a SHOW MEASURES query.

measure_name	data_type	Dimensions
dashboard-metrics	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]

Example: Using measure name from scheduled query in multi-measure records

In this example, you will see a query reading from a table with single-measure records and materializing the results into multi-measure records. In this case, the scheduled query result has a column whose values can be used as measure names in the target table where the results of the scheduled query is materialized. Then you can specify the measure name for the multi-measure record in the derived table using the MeasureNameColumn property in TargetConfiguration.TimestreamConfiguration.

```
{
    "Name" : "UsingMeasureNameFromQueryResult",
    "QueryString" : "SELECT region, bin(time, 1h) as hour, measure_name, AVG(CASE WHEN
 measure_name IN ('memory_cached', 'disk_used', 'disk_free') THEN measure_value::double
 ELSE NULL END) as avg_1h, MIN(CASE WHEN measure_name IN ('memory_free', 'cpu_idle')
 THEN measure_value::double ELSE NULL END) as min_1h, SUM(CASE WHEN measure_name
 IN ('disk_io_writes') THEN measure_value::double ELSE NULL END) as sum_1h,
 MAX(CASE WHEN measure_name IN ('memory_used', 'cpu_user', 'cpu_system') THEN
 measure_value::double ELSE NULL END) as max_1h FROM raw_data.devops WHERE time
 BETWEEN bin(@scheduled_runtime, 1h) - 14h AND bin(@scheduled_runtime, 1h) - 2h AND
 measure_name IN ('memory_free', 'memory_used', 'memory_cached', 'disk_io_writes',
 'disk_used', 'disk_free', 'cpu_user', 'cpu_system', 'cpu_idle') GROUP BY region,
 measure_name, bin(time, 1h)",
    "ScheduleConfiguration" : {
        "ScheduleExpression" : "cron(0 0/1 * * ? *)"
    },
    "NotificationConfiguration" : {
        "SnsConfiguration" : {
            "TopicArn" : "******"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******",
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName" : "derived",
            "TableName" : "dashboard_metrics_1h_agg_2",
```

```
            "TimeColumn" : "hour",
            "DimensionMappings" : [
                {
                    "Name": "region",
                    "DimensionValueType" : "VARCHAR"
                }
            ],
            "MeasureNameColumn" : "measure_name",
            "MultiMeasureMappings" : {
                "MultiMeasureAttributeMappings" : [
                    {
                        "SourceColumn" : "avg_1h",
                        "MeasureValueType" : "DOUBLE"
                    },
                    {
                        "SourceColumn" : "min_1h",
                        "MeasureValueType" : "DOUBLE",
                        "TargetMultiMeasureAttributeName": "p0_1h"
                    },
                    {
                        "SourceColumn" : "sum_1h",
                        "MeasureValueType" : "DOUBLE"
                    },
                    {
                        "SourceColumn" : "max_1h",
                        "MeasureValueType" : "DOUBLE",
                        "TargetMultiMeasureAttributeName": "p100_1h"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    }
}
```

The mapping in this example will create multi-measure records with attributes avg_1h, p0_1h, sum_1h, p100_1h and will use the values of the measure_name column in the query result as

the measure name for the multi-measure records in the destination table. Additionally note that the previous examples optionally use the TargetMultiMeasureAttributeName with a subset of the mappings to rename the attributes. For instance, min_1h was renamed to p0_1h and max_1h is renamed to p100_1h.

The following is the schema for the destination table once this scheduled query is materialized. As you can see from the Timestream for LiveAnalytics attribute type in the following result, the results are materialized into a multi-measure record. If you look at the measure schema, there were nine different measure names that were ingested which correspond to the values seen in the query results.

Column	Type	Timestream for LiveAnalytics attribute type
region	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
sum_1h	double	MULTI
p100_1h	double	MULTI
p0_1h	double	MULTI
avg_1h	double	MULTI

The following are corresponding measures obtained with a SHOW MEASURES query.

measure_name	data_type	Dimensions
cpu_idle	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
cpu_system	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
cpu_user	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
disk_free	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
disk_io_writes	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
disk_used	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
memory_cached	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
memory_free	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
memory_free	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]

Example: Mapping results to different multi-measure records with different attributes

The following example shows how you can map different columns in your query result into different multi-measure records with different measure names. If you see the following scheduled query definition, the result of the query has the following columns: region, hour, avg_mem_cached_1h, min_mem_free_1h, max_mem_used_1h, total_disk_io_writes_1h, avg_disk_used_1h, avg_disk_free_1h, max_cpu_user_1h, max_cpu_system_1h, min_cpu_system_1h. `region` is mapped to dimension, and `hour` is mapped to the time column.

The MixedMeasureMappings property in TargetConfiguration.TimestreamConfiguration specifies how to map the measures to multi-measure records in the derived table.

In this specific example, avg_mem_cached_1h, min_mem_free_1h, max_mem_used_1h are used in one multi-measure record with measure name of mem_aggregates, total_disk_io_writes_1h, avg_disk_used_1h, avg_disk_free_1h are used in another multi-measure record with measure name

of disk_aggregates, and finally max_cpu_user_1h, max_cpu_system_1h, min_cpu_system_1h are used in another multi-measure record with measure name cpu_aggregates.

In these mappings, you can also optionally use TargetMultiMeasureAttributeName to rename the query result column to have a different attribute name in the destination table. For instance, the result column avg_mem_cached_1h gets renamed to avgMemCached, total_disk_io_writes_1h gets renamed to totalIOWrites, etc.

When you're defining the mappings for multi-measure records, Timestream for LiveAnalytics inspects every row in the query results and automatically ignores the column values that have NULL values. As a result, in the case of mappings with multiple measures names, if all the column values for that group in the mapping are NULL for a given row, then no value for that measure name is ingested for that row.

For example, in the following mapping, avg_mem_cached_1h, min_mem_free_1h, and max_mem_used_1h are mapped to measure name mem_aggregates. If for a given row of the query result, all these of the column values are NULL, Timestream for LiveAnalytics won't ingest the measure mem_aggregates for that row. If all nine columns for a given row are NULL, then you will see an user error reported in your error report.

```
{
    "Name" : "AggsInDifferentMultiMeasureRecords",
    "QueryString" : "SELECT region, bin(time, 1h) as hour, AVG(CASE WHEN measure_name
 = 'memory_cached' THEN measure_value::double ELSE NULL END) as avg_mem_cached_1h,
 MIN(CASE WHEN measure_name = 'memory_free' THEN measure_value::double ELSE
 NULL END) as min_mem_free_1h, MAX(CASE WHEN measure_name = 'memory_used' THEN
 measure_value::double ELSE NULL END) as max_mem_used_1h, SUM(CASE WHEN measure_name =
 'disk_io_writes' THEN measure_value::double ELSE NULL END) as total_disk_io_writes_1h,
 AVG(CASE WHEN measure_name = 'disk_used' THEN measure_value::double ELSE NULL END) as
 avg_disk_used_1h, AVG(CASE WHEN measure_name = 'disk_free' THEN measure_value::double
 ELSE NULL END) as avg_disk_free_1h, MAX(CASE WHEN measure_name = 'cpu_user' THEN
 measure_value::double ELSE NULL END) as max_cpu_user_1h, MAX(CASE WHEN measure_name
 = 'cpu_system' THEN measure_value::double ELSE NULL END) as max_cpu_system_1h,
 MIN(CASE WHEN measure_name = 'cpu_idle' THEN measure_value::double ELSE NULL END)
 as min_cpu_system_1h FROM raw_data.devops WHERE time BETWEEN bin(@scheduled_runtime,
 1h) - 14h AND bin(@scheduled_runtime, 1h) - 2h AND measure_name IN ('memory_cached',
 'memory_free', 'memory_used', 'disk_io_writes', 'disk_used', 'disk_free', 'cpu_user',
 'cpu_system', 'cpu_idle') GROUP BY region, bin(time, 1h)",
    "ScheduleConfiguration" : {
        "ScheduleExpression" : "cron(0 0/1 * * ? *)"
    },
    "NotificationConfiguration" : {
```

```
        "SnsConfiguration" : {
            "TopicArn" : "******"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******",
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName" : "derived",
            "TableName" : "dashboard_metrics_1h_agg_3",
            "TimeColumn" : "hour",
            "DimensionMappings" : [
                {
                    "Name": "region",
                    "DimensionValueType" : "VARCHAR"
                }
            ],
            "MixedMeasureMappings" : [
                {
                    "MeasureValueType" : "MULTI",
                    "TargetMeasureName" : "mem_aggregates",
                    "MultiMeasureAttributeMappings" : [
                        {
                            "SourceColumn" : "avg_mem_cached_1h",
                            "MeasureValueType" : "DOUBLE",
                            "TargetMultiMeasureAttributeName": "avgMemCached"
                        },
                        {
                            "SourceColumn" : "min_mem_free_1h",
                            "MeasureValueType" : "DOUBLE"
                        },
                        {
                            "SourceColumn" : "max_mem_used_1h",
                            "MeasureValueType" : "DOUBLE",
                            "TargetMultiMeasureAttributeName": "maxMemUsed"
                        }
                    ]
                },
                {
                    "MeasureValueType" : "MULTI",
                    "TargetMeasureName" : "disk_aggregates",
                    "MultiMeasureAttributeMappings" : [
                        {
                            "SourceColumn" : "total_disk_io_writes_1h",
                            "MeasureValueType" : "DOUBLE",
```

```
                            "TargetMultiMeasureAttributeName": "totalIOWrites"
                        },
                        {
                            "SourceColumn" : "avg_disk_used_1h",
                            "MeasureValueType" : "DOUBLE"
                        },
                        {
                            "SourceColumn" : "avg_disk_free_1h",
                            "MeasureValueType" : "DOUBLE"
                        }
                    ]
                },
                {
                    "MeasureValueType" : "MULTI",
                    "TargetMeasureName" : "cpu_aggregates",
                    "MultiMeasureAttributeMappings" : [
                        {
                            "SourceColumn" : "max_cpu_user_1h",
                            "MeasureValueType" : "DOUBLE"
                        },
                        {
                            "SourceColumn" : "max_cpu_system_1h",
                            "MeasureValueType" : "DOUBLE"
                        },
                        {
                            "SourceColumn" : "min_cpu_idle_1h",
                            "MeasureValueType" : "DOUBLE",
                            "TargetMultiMeasureAttributeName": "minCpuIdle"
                        }
                    ]
                }
            ]
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    }
}
```

The following is the schema for the destination table once this scheduled query is materialized.

Column	Type	Timestream for LiveAnalytics attribute type
region	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
minCpuIdle	double	MULTI
max_cpu_system_1h	double	MULTI
max_cpu_user_1h	double	MULTI
avgMemCached	double	MULTI
maxMemUsed	double	MULTI
min_mem_free_1h	double	MULTI
avg_disk_free_1h	double	MULTI
avg_disk_used_1h	double	MULTI
totalIOWrites	double	MULTI

The following are the corresponding measures obtained with a SHOW MEASURES query.

measure_name	data_type	Dimensions
cpu_aggregates	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]
disk_aggregates	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
mem_aggregates	multi	[{'dimension_name': 'region', 'data_type': 'varchar'}]

Example: Mapping results to single-measure records with measure name from query results

The following is an example of a scheduled query whose results are materialized into single-measure records. In this example, the query result has the measure_name column whose values will be used as measure names in the target table. You use the MixedMeasureMappings attribute in the TargetConfiguration.TimestreamConfiguration to specify the mapping of the query result column to the scalar measure in the target table.

In the following example definition, the query result is expected to nine distinct measure_name values. You list out all these measure names in the mapping and specify which column to use for the single-measure value for that measure name. For example, in this mapping, if measure name of memory_cached is seen for a given result row, then the value in the avg_1h column is used as the value for the measure when the data is written to the target table. You can optionally use TargetMeasureName to provide a new measure name for this value.

```
{
    "Name" : "UsingMeasureNameColumnForSingleMeasureMapping",
    "QueryString" : "SELECT region, bin(time, 1h) as hour, measure_name, AVG(CASE WHEN
 measure_name IN ('memory_cached', 'disk_used', 'disk_free') THEN measure_value::double
 ELSE NULL END) as avg_1h, MIN(CASE WHEN measure_name IN ('memory_free', 'cpu_idle')
 THEN measure_value::double ELSE NULL END) as min_1h, SUM(CASE WHEN measure_name
 IN ('disk_io_writes') THEN measure_value::double ELSE NULL END) as sum_1h,
 MAX(CASE WHEN measure_name IN ('memory_used', 'cpu_user', 'cpu_system') THEN
 measure_value::double ELSE NULL END) as max_1h FROM raw_data.devops WHERE time
 BETWEEN bin(@scheduled_runtime, 1h) - 14h AND bin(@scheduled_runtime, 1h) - 2h AND
 measure_name IN ('memory_free', 'memory_used', 'memory_cached', 'disk_io_writes',
 'disk_used', 'disk_free', 'cpu_user', 'cpu_system', 'cpu_idle') GROUP BY region,
 bin(time, 1h), measure_name",
    "ScheduleConfiguration" : {
        "ScheduleExpression" : "cron(0 0/1 * * ? *)"
    },
    "NotificationConfiguration" : {
        "SnsConfiguration" : {
            "TopicArn" : "******"
```

```
        }
    },
    "ScheduledQueryExecutionRoleArn": "******",
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName" : "derived",
            "TableName" : "dashboard_metrics_1h_agg_4",
            "TimeColumn" : "hour",
            "DimensionMappings" : [
                {
                    "Name": "region",
                    "DimensionValueType" : "VARCHAR"
                }
            ],
            "MeasureNameColumn" : "measure_name",
            "MixedMeasureMappings" : [
                {
                    "MeasureName" : "memory_cached",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "avg_1h",
                    "TargetMeasureName" : "AvgMemCached"
                },
                {
                    "MeasureName" : "disk_used",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "avg_1h"
                },
                {
                    "MeasureName" : "disk_free",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "avg_1h"
                },
                {
                    "MeasureName" : "memory_free",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "min_1h",
                    "TargetMeasureName" : "MinMemFree"
                },
                {
                    "MeasureName" : "cpu_idle",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "min_1h"
                },
                {
```

```
                    "MeasureName" : "disk_io_writes",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "sum_1h",
                    "TargetMeasureName" : "total-disk-io-writes"
                },
                {
                    "MeasureName" : "memory_used",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "max_1h",
                    "TargetMeasureName" : "maxMemUsed"
                },
                {
                    "MeasureName" : "cpu_user",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "max_1h"
                },
                {
                    "MeasureName" : "cpu_system",
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "max_1h"
                }
            ]
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    }
}
```

The following is the schema for the destination table once this scheduled query is materialized. As you can see from the schema, the table is using single-measure records. If you list the measure schema for the table, you will see the nine measures written to based on the mapping provided in the specification.

Column	Type	Timestream for LiveAnalytics attribute type
region	varchar	DIMENSION

Column	Type	Timestream for LiveAnalytics attribute type
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
measure_value::double	double	MEASURE_VALUE

The following are the corresponding measures obtained with a SHOW MEASURES query.

measure_name	data_type	Dimensions
AvgMemCached	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
MinMemFree	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
cpu_idle	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
cpu_system	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
cpu_user	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
disk_free	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
disk_used	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
maxMemUsed	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
total-disk-io-writes	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]

Example: Mapping results to single-measure records with query result columns as measure names

In this example, you have a query whose results do not have a measure name column. Instead, you want the query result column name as the measure name when mapping the output to single-measure records. Earlier there was an example where a similar result was written to a multi-measure record. In this example, you will see how to map it to single-measure records if that fits your application scenario.

Again, you specify this mapping using the MixedMeasureMappings property in TargetConfiguration.TimestreamConfiguration. In the following example, you see that the query result has nine columns. You use the result columns as measure names and the values as the single-measure values.

For example, for a given row in the query result, the column name avg_mem_cached_1h is used as the column name and value associated with column, and avg_mem_cached_1h is used as the measure value for the single-measure record. You can also use TargetMeasureName to use a different measure name in the target table. For instance, for values in column sum_1h, the mapping specifies to use total_disk_io_writes_1h as the measure name in the target table. If any column's value is NULL, then the corresponding measure is ignored.

```
{
    "Name" : "SingleMeasureMappingWithoutMeasureNameColumnInQueryResult",
    "QueryString" : "SELECT region, bin(time, 1h) as hour, AVG(CASE WHEN measure_name
 = 'memory_cached' THEN measure_value::double ELSE NULL END) as avg_mem_cached_1h,
 AVG(CASE WHEN measure_name = 'disk_used' THEN measure_value::double ELSE NULL END) as
 avg_disk_used_1h, AVG(CASE WHEN measure_name = 'disk_free' THEN measure_value::double
 ELSE NULL END) as avg_disk_free_1h, MIN(CASE WHEN measure_name = 'memory_free' THEN
 measure_value::double ELSE NULL END) as min_mem_free_1h, MIN(CASE WHEN measure_name =
 'cpu_idle' THEN measure_value::double ELSE NULL END) as min_cpu_idle_1h, SUM(CASE WHEN
 measure_name = 'disk_io_writes' THEN measure_value::double ELSE NULL END) as sum_1h,
 MAX(CASE WHEN measure_name = 'memory_used' THEN measure_value::double ELSE NULL END)
 as max_mem_used_1h, MAX(CASE WHEN measure_name = 'cpu_user' THEN measure_value::double
 ELSE NULL END) as max_cpu_user_1h, MAX(CASE WHEN measure_name = 'cpu_system' THEN
 measure_value::double ELSE NULL END) as max_cpu_system_1h FROM raw_data.devops WHERE
 time BETWEEN bin(@scheduled_runtime, 1h) - 14h AND bin(@scheduled_runtime, 1h) - 2h
 AND measure_name IN ('memory_free', 'memory_used', 'memory_cached', 'disk_io_writes',
 'disk_used', 'disk_free', 'cpu_user', 'cpu_system', 'cpu_idle') GROUP BY region,
 bin(time, 1h)",
    "ScheduleConfiguration" : {
        "ScheduleExpression" : "cron(0 0/1 * * ? *)"
```

```
    },
    "NotificationConfiguration" : {
        "SnsConfiguration" : {
            "TopicArn" : "******"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******",
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName" : "derived",
            "TableName" : "dashboard_metrics_1h_agg_5",
            "TimeColumn" : "hour",
            "DimensionMappings" : [
                {
                    "Name": "region",
                    "DimensionValueType" : "VARCHAR"
                }
            ],
            "MixedMeasureMappings" : [
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "avg_mem_cached_1h"
                },
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "avg_disk_used_1h"
                },
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "avg_disk_free_1h"
                },
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "min_mem_free_1h"
                },
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "min_cpu_idle_1h"
                },
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "sum_1h",
                    "TargetMeasureName" : "total_disk_io_writes_1h"
                },
```

```
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "max_mem_used_1h"
                },
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "max_cpu_user_1h"
                },
                {
                    "MeasureValueType" : "DOUBLE",
                    "SourceColumn" : "max_cpu_system_1h"
                }
            ]
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    }
}
```

The following is the schema for the destination table once this scheduled query is materialized. As you can see that the target table is storing records with single-measure values of type double. Similarly, the measure schema for the table shows the nine measure names. Also notice that the measure name total_disk_io_writes_1h is present since the mapping renamed sum_1h to total_disk_io_writes_1h.

Column	Type	Timestream for LiveAnalytics attribute type
region	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
measure_value::double	double	MEASURE_VALUE

The following are the corresponding measures obtained with a SHOW MEASURES query.

measure_name	data_type	Dimensions
avg_disk_free_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
avg_disk_used_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
avg_mem_cached_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
max_cpu_system_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
max_cpu_user_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
max_mem_used_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
min_cpu_idle_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
min_mem_free_1h	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]
total-disk-io-writes	double	[{'dimension_name': 'region', 'data_type': 'varchar'}]

Scheduled query notification messages

This section describes the messages sent by Timestream for LiveAnalytics when creating, deleting, running, or updating the state of a scheduled query.

Notification message name	Structure	Description
CreatingNotificationMessage	`CreatingNotificationMessage { String arn; NotificationType type; }`	This notification message is sent before sending the response for `CreateScheduledQuery`. The scheduled query is enabled after sending this notification. *arn* - The ARN of the scheduled query that is being created. *type* - SCHEDULED_QUERY_CREATING
UpdateNotificationMessage	`UpdateNotificationMessage { String arn; NotificationType type; QueryState state; }`	This notification message is sent when a scheduled query is updated. Timestream for LiveAnalytics can disable the scheduled query, automatically, in case non-recoverable error is encountered, such as: • AssumeRole failure • Any 4xx errors encountered when communicating with KMS when a customer managed KMS key is specified. • Any 4xx errors encountered during running of the scheduled query. • Any 4xx errors encountered during ingestion of query results

Notification message name	Structure	Description
		arn - The ARN of the scheduled query that is being updated. *type* - SCHEDULED_QUERY_UPDATE *state* - ENABLED or DISABLED
DeleteNotificationMessage	`DeletionNotificationMessage { String arn; NotificationType type; }`	This notification message is sent when a scheduled query has been deleted. *arn* - The ARN of the scheduled query that is being created. *type* - SCHEDULED_QUERY_DELETED

Notification message name	Structure	Description

SuccessNotificationMessage	`SuccessNotificationMessage {`
	`  NotificationType type;`
	`  String arn;`
	`  Date nextInvocationEpochSecond;`
	`  ScheduledQueryRunSummary runSummary;`
	`}`
	
	`ScheduledQueryRunSummary {`
	`  Date invocationTime;`
	`  Date triggerTime;`
	`  String runStatus;`
	`  ExecutionStats executionstats;`
	`  ErrorReportLocation errorReportLocation;`
	`  String failureReason;`
	`}`
	
	`ExecutionStats {`
	`  Long bytesMetered;`
	`  Long dataWrites;`
	`  Long queryResultRows;`
	`  Long recordsIngested;`
	`  Long executionTimeInMillis;`
	`}`
	
	`ErrorReportLocation {`
		This notification message is sent after the scheduled query is run and the results are successfully ingested.
		
		ARN - The ARN of the scheduled query that is being deleted.
		
		NotificationType - AUTO_TRIGGER_SUCCESS or MANUAL_TRIGGER_SUCCESS.
		
		nextInvocationEpochSecond - The next time Timestream for LiveAnalytics will run the scheduled query.
		
		runSummary - Information about the scheduled query run.

Notification message name	Structure	Description

	`    S3ReportLocation s3ReportLocation;`
	`}`
	
	`S3ReportLocation {`
	`    String bucketName;`
	`    String objectKey;`
	`}`	

Notification message name	Structure	Description

FailureNotificationMessage	`FailureNotificationMessage {`
	`  NotificationType type;`
	`  String arn;`
	`  ScheduledQueryRunSummary runSummary;`
	`}`
	
	`ScheduledQueryRunSummary {`
	`  Date invocationTime;`
	`  Date triggerTime;`
	`  String runStatus;`
	`  ExecutionStats executionstats;`
	`  ErrorReportLocation errorReportLocation;`
	`  String failureReason;`
	`}`
	
	`ExecutionStats {`
	`  Long bytesMetered;`
	`  Long dataWrites;`
	`  Long queryResultRows;`
	`  Long recordsIngested;`
	`  Long executionTimeInMillis;`
	`}`
	
	`ErrorReportLocation {`
	`  S3ReportLocation s3ReportLocation;`
	`}`
		This notification message is sent when a failure is encountered during a scheduled query run or when ingesting the query results.
		
		arn - The ARN of the scheduled query that is being run.
		
		type - AUTO_TRIGGER_FAILURE or MANUAL_TRIGGER_FAILURE.
		
		runSummary - Information about the scheduled query run.

Notification message name	Structure	Description
	`S3ReportLocation { String bucketName; String objectKey; }`	

Scheduled query error reports

This section describes the location, format, and reasons for error reports generated by Timestream for LiveAnalytics when errors are encountered by running scheduled queries.

Topics

- Scheduled query error reports reasons
- Scheduled query error reports location
- Scheduled query error reports format
- Scheduled query error types
- Scheduled query error reports example

Scheduled query error reports reasons

Error reports are generated for recoverable errors. Error reports are not generated for non-recoverable errors. Timestream for LiveAnalytics can disable the scheduled queries automatically when non-recoverable errors are encountered. These include:

- `AssumeRole` failure
- Any 4xx errors encountered when communicating with KMS when a customer-managed KMS key is specified
- Any 4xx errors encountered when a scheduled query runs
- Any 4xx errors encountered during ingestion of query results

For non-recoverable errors, Timestream for LiveAnalytics sends a failure notification with a non-recoverable error message. An update notification is also sent which indicates that the scheduled query is disabled.

Scheduled query error reports location

A scheduled query error report location has the following naming convention:

```
s3://customer-bucket/customer-prefix/
```

Following is an example scheduled query ARN:

```
arn:aws:timestream:us-east-1:000000000000:scheduled-query/test-query-hd734tegrgfd
```

```
s3://customer-bucket/customer-prefix/test-query-hd734tegrgfd/<InvocationTime>/<Auto or
 Manual>/<Actual Trigger Time>
```

Auto indicates scheduled queries automatically scheduled by Timestream for LiveAnalytics and *Manual* indicates scheduled queries manually triggered by a user via `ExecuteScheduledQuery` API action in Amazon Timestream for LiveAnalytics Query. For more information about `ExecuteScheduledQuery`, see ExecuteScheduledQuery.

Scheduled query error reports format

The error reports have the following JSON format:

```
{
    "reportId": <String>,            // A unique string ID for all error reports
 belonging to a particular scheduled query run
    "errors": [ <Error>, ... ],      // One or more errors
}
```

Scheduled query error types

The `Error` object can be one of three types:

- Records Ingestion Errors

```
{
    "reason": <String>,              // The error message String
    "records": [ <Record>, ... ],    // One or more rejected records )
```

```
}
```

- Row Parse and Validation Errors

```
{
    "reason": <String>,        // The error message String
    "rawLine": <String>,       // [Optional] The raw line String that is being parsed
 into record(s) to be ingested. This line has encountered the above-mentioned parse
 error.
}
```

- General Errors

```
{
    "reason": <String>,        // The error message
}
```

Scheduled query error reports example

The following is an example of an error report that was produced due to ingestion errors.

```
{
    "reportId": "C9494AABE012D1FBC162A67EA2C18255",
    "errors": [
        {
            "reason": "The record timestamp is outside the time range
 [2021-11-12T14:18:13.354Z, 2021-11-12T16:58:13.354Z) of the memory store.",
            "records": [
                {
                    "dimensions": [
                        {
                            "name": "dim0",
                            "value": "d0_1",
                            "dimensionValueType": null
                        },
                        {
                            "name": "dim1",
                            "value": "d1_1",
                            "dimensionValueType": null
                        }
                    ],
                    "measureName": "random_measure_value",
```

```
                "measureValue": "3.141592653589793",
                "measureValues": null,
                "measureValueType": "DOUBLE",
                "time": "1637166175635000000",
                "timeUnit": "NANOSECONDS",
                "version": null
            },
            {
                "dimensions": [
                    {
                        "name": "dim0",
                        "value": "d0_2",
                        "dimensionValueType": null
                    },
                    {
                        "name": "dim1",
                        "value": "d1_2",
                        "dimensionValueType": null
                    }
                ],
                "measureName": "random_measure_value",
                "measureValue": "6.283185307179586",
                "measureValues": null,
                "measureValueType": "DOUBLE",
                "time": "1637166175636000000",
                "timeUnit": "NANOSECONDS",
                "version": null
            },
            {
                "dimensions": [
                    {
                        "name": "dim0",
                        "value": "d0_3",
                        "dimensionValueType": null
                    },
                    {
                        "name": "dim1",
                        "value": "d1_3",
                        "dimensionValueType": null
                    }
                ],
                "measureName": "random_measure_value",
                "measureValue": "9.42477796076938",
                "measureValues": null,
```

```
                    "measureValueType": "DOUBLE",
                    "time": "1637166175637000000",
                    "timeUnit": "NANOSECONDS",
                    "version": null
                },
                {
                    "dimensions": [
                        {
                            "name": "dim0",
                            "value": "d0_4",
                            "dimensionValueType": null
                        },
                        {
                            "name": "dim1",
                            "value": "d1_4",
                            "dimensionValueType": null
                        }
                    ],
                    "measureName": "random_measure_value",
                    "measureValue": "12.566370614359172",
                    "measureValues": null,
                    "measureValueType": "DOUBLE",
                    "time": "1637166175638000000",
                    "timeUnit": "NANOSECONDS",
                    "version": null
                }
            ]
        }
    ]
}
```


Scheduled query patterns and examples

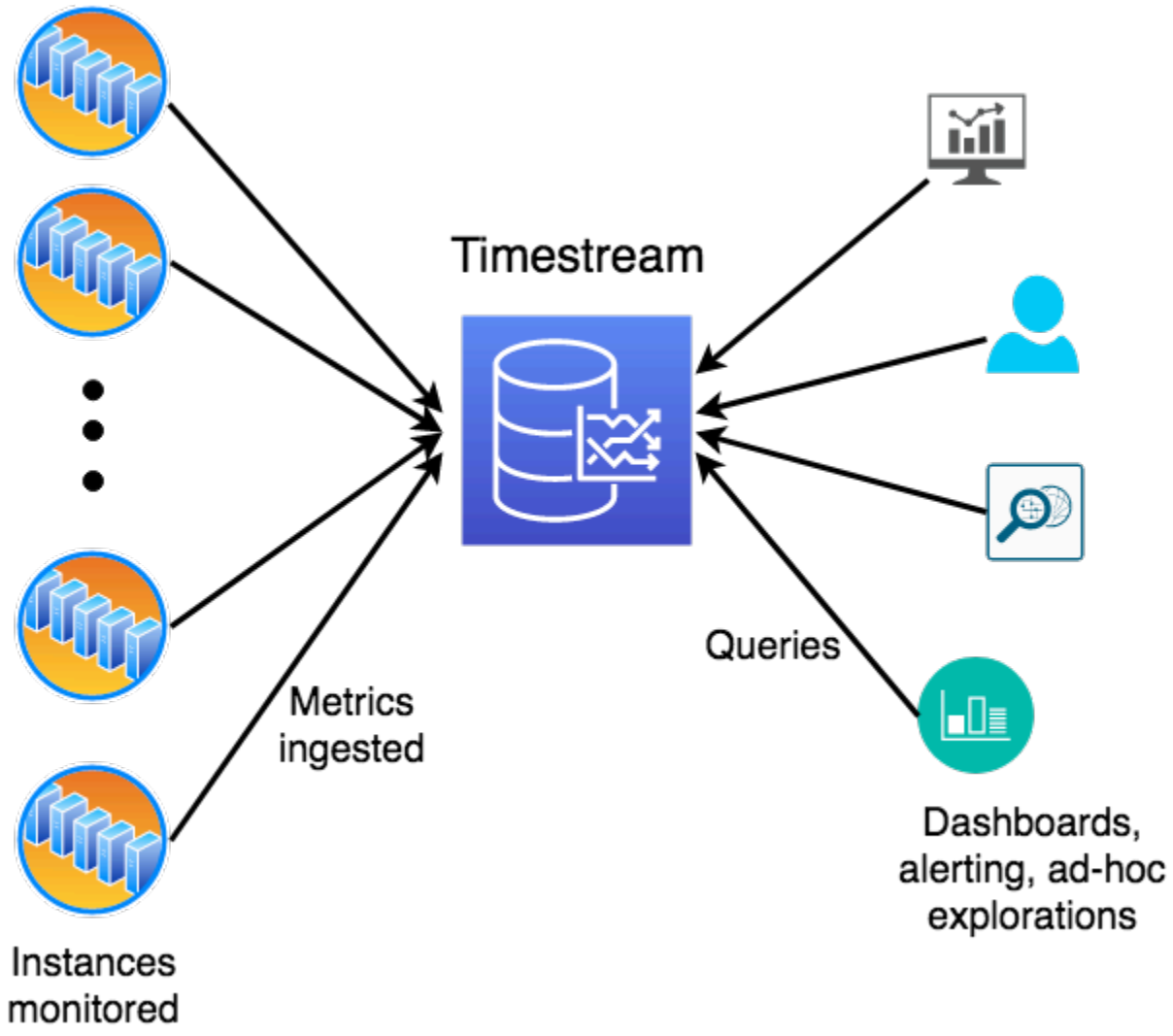
This section describes the usage patterns for scheduled queries as well as end-to-end examples.

Topics

- Scheduled queries sample schema
- Scheduled query patterns
- Scheduled query examples

Scheduled queries sample schema

In this example we will use a sample application mimicking a DevOps scenario monitoring metrics from a large fleet of servers. Users want to alert on anomalous resource usage, create dashboards on aggregate fleet behavior and utilization, and perform sophisticated analysis on recent and historical data to find correlations. The following diagram provides an illustration of the setup where a set of monitored instances emit metrics to Timestream for LiveAnalytics. Another set of concurrent users issues queries for alerts, dashboards, or ad-hoc analysis, where queries and ingestion run in parallel.



The application being monitored is modeled as a highly scaled-out service that is deployed in several regions across the globe. Each region is further subdivided into a number of scaling units called cells that have a level of isolation in terms of infrastructure within the region. Each cell is further subdivided into silos, which represent a level of software isolation. Each silo has five

microservices that comprise one isolated instance of the service. Each microservice has several servers with different instance types and OS versions, which are deployed across three availability zones. These attributes that identify the servers emitting the metrics are modeled as dimensions in Timestream for LiveAnalytics. In this architecture, we have a hierarchy of dimensions (such as region, cell, silo, and microservice_name) and other dimensions that cut across the hierarchy (such as instance_type and availability_zone).

The application emits a variety of metrics (such as cpu_user and memory_free) and events (such as task_completed and gc_reclaimed). Each metric or event is associated with eight dimensions (such as region or cell) that uniquely identify the server emitting it. Data is written with the 20 metrics stored together in a multi-measure record with measure name metrics and all the 5 events are stored together in another multi-measure record with measure name events. The data model, schema, and data generation can be found in the open-sourced data generator. In addition to the schema and data distributions, the data generator provides an example of using multiple writers to ingest data in parallel, using the ingestion scaling of Timestream for LiveAnalytics to ingest millions of measurements per second. Below we show the schema (table and measure schema) and some sample data from the data set.

Topics

- Multi-measure records
- Single-measure records

Multi-measure records

Table Schema

Below is the table schema once the data is ingested using multi-measure records. It is the output of DESCRIBE query. Assuming the data is ingested into a database raw_data and table devops, below is the query.

```
DESCRIBE "raw_data"."devops"
```

Column	Type	Timestream for LiveAnalytics attribute type
availability_zone	varchar	DIMENSION

Column	Type	Timestream for LiveAnalytics attribute type
microservice_name	varchar	DIMENSION
instance_name	varchar	DIMENSION
process_name	varchar	DIMENSION
os_version	varchar	DIMENSION
jdk_version	varchar	DIMENSION
cell	varchar	DIMENSION
region	varchar	DIMENSION
silo	varchar	DIMENSION
instance_type	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
memory_free	double	MULTI
cpu_steal	double	MULTI
cpu_iowait	double	MULTI
cpu_user	double	MULTI
memory_cached	double	MULTI
disk_io_reads	double	MULTI
cpu_hi	double	MULTI
latency_per_read	double	MULTI
network_bytes_out	double	MULTI

Column	Type	Timestream for LiveAnalytics attribute type
cpu_idle	double	MULTI
disk_free	double	MULTI
memory_used	double	MULTI
cpu_system	double	MULTI
file_descriptors_in_use	double	MULTI
disk_used	double	MULTI
cpu_nice	double	MULTI
disk_io_writes	double	MULTI
cpu_si	double	MULTI
latency_per_write	double	MULTI
network_bytes_in	double	MULTI
task_end_state	varchar	MULTI
gc_pause	double	MULTI
task_completed	bigint	MULTI
gc_reclaimed	double	MULTI

Measure Schema

Below is the measure schema returned by the SHOW MEASURES query.

```
SHOW MEASURES FROM "raw_data"."devops"
```

measure_name	data_type	Dimensions
events	multi	[{"data_type":"varchar","dimension_name":"availability_zone"},{"data_type":"varchar","dimension_name":"microservice_name"},{"data_type":"varchar","dimension_name":"instance_name"},{"data_type":"varchar","dimension_name":"process_name"},{"data_type":"varchar","dimension_name":"jdk_version"},{"data_type":"varchar","dimension_name":"cell"},{"data_type":"varchar","dimension_name":"region"},{"data_type":"varchar","dimension_name":"silo"}]
metrics	multi	[{"data_type":"varchar","dimension_name":"availability_zone"},{"data_type":"varchar","dimension_name":"microservice_name"},{"data_type":"varchar","dimension_name":"instance_name"},{"data_type":"varchar","dimension_name":"os_version"},{"data_type":"varchar","dimension_name":"cell"},{"data_type":"varchar","dimension_name":"region"},{"data_type":"varchar","dimension_name":"silo"},{"data_type":"varchar","d

measure_name	data_type	Dimensions
		imension_name":"instance_type"}]

Example Data

re	C	S	a	m	in	in	o	p	jo	m	T	c	c	c	c	c	c	c	c	m	m	d	la	d	la	d	d	n	n	fi	m	ta	g	ta	g	s	e	d
u	u	u	u	a	i-	m	A			m	1	6	0	3	0	0	0	0	0	5	8	5	9	3	2	6	2	8	4	3	5							
e	e	e	e		za	e					1																											
	-	-	-		k-						1																											
	ce	ce	-1		at																																	
		s			u																																	
		il			e																																	
					-																																	
					ce																																	
					s																																	
					il																																	
					0																																	
					zo																																	
					m																																	
u	u	u	u	a	i-	m	A			m	1	5	0	3	0	0	0	0	0	9	3	5	3	2	9	5	3	7	5	6	2							
e	e	e	e		za	e					1																											
	-	-	-1		k-						1																											
	ce	ce			at																																	
		s			u																																	
		il			e																																	
					-																																	
					ce																																	
					s																																	
					il																																	
					0																																	

re	C	S	a it	m ic	in n	in ty	o n	p a	jo o	m a	T	c	c m	c	c	c t	c	c	c	m e	m cl	d e	la e	d ri	la e	d	d	n y	n y	fi ri n	m e	ta st	g	ta le	gs ed	se d
					zo m																															
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 zo m	m e	A			m	1 1 1	4	0	4	0	0	0	0	0	9	4	5	3	8	3	7	6	8	5	4	8				
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 zo m	m e	A			m	1 1 1	3	0	5	0	0	0	0	0	4	7	2	4	4	3	8	6	2	1	2	1				

re	C	S	a it	m ic	in n	in ty	o n	p a	jo o	m a	T	c	c m	c	c	c t	c	c	c	m e	m cl	d e	la e	d ri	la e	d	d	n y	n y	fi ri n	m e	ta st	g	ta le	gs ed	se d	p m d
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 z m	m e	A			m	1 1 1	5	0	3	0	0	0	0	0	5	3	8	5	7	1	8	6	7	6	5	3						
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 z m			h g	JI	e	1 1 1																				7	S IT S	8	3	64	AW RE		

re	C	S	a it	m ic	in n	in ty	o n	p a	jo o	m a	T	c	c m	c	c	c t	c	c	c	m e	m cl	d e	la e	d ri	la e	d	d	n y	n y	fi ri n	m e	ta st	g	ta le	g m	se ed	p ed
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 z m			h g	JI	e	1 1 1																				7	S IT T	2	4	7	4 IL	5
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 z m			h g	JI	e	1 1 1																				6	S IT T	6	2	7	2 IL	3

re	C	S	a it	m ic	in n	in ty	o n	p a	jo o	m a	T	c	c m	c	c	c t	c	c	c	m e	m cl	d e	la e	d ri	la e	d	d	n y	n y	fi ri n	m e	ta st	g	ta le	gs m	se ed	p ed
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 z m			h g	JI	e	1 1 1																				2	S IT T	5	1	99	W IL	
u e	u e - c	u e - c s il	u e -	a	i- z k- a u e - c s il 0 z m			h g	JI	e	1 1 1																				3	S IT S	7	2	82	20 RE	

Single-measure records

Timestream for LiveAnalytics also allows you to ingest the data with one measure per time series record. Below are the schema details when ingested using single measure records.

Table Schema

Below is the table schema once the data is ingested using multi-measure records. It is the output of DESCRIBE query. Assuming the data is ingested into a database raw_data and table devops, below is the query.

```
DESCRIBE "raw_data"."devops_single"
```

Column	Type	Timestream for LiveAnalytics attribute type
availability_zone	varchar	DIMENSION
microservice_name	varchar	DIMENSION
instance_name	varchar	DIMENSION
process_name	varchar	DIMENSION
os_version	varchar	DIMENSION
jdk_version	varchar	DIMENSION
cell	varchar	DIMENSION
region	varchar	DIMENSION
silo	varchar	DIMENSION
instance_type	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
measure_value::double	double	MEASURE_VALUE
measure_value::bigint	bigint	MEASURE_VALUE
measure_value::varchar	varchar	MEASURE_VALUE

Measure Schema

Below is the measure schema returned by the SHOW MEASURES query.

```
SHOW MEASURES FROM "raw_data"."devops_single"
```

measure_name	data_type	Dimensions
cpu_hi	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]
cpu_idle	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'},

measure_name	data_type	Dimensions
		{'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]
cpu_iowait	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
cpu_nice	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
cpu_si	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
cpu_steal	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
cpu_system	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
cpu_user	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
disk_free	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
disk_io_reads	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
disk_io_writes	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
disk_used	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
file_descriptors_in_use	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
gc_pause	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'process_name', 'data_type': 'varchar'}, {'dimension_name': 'jdk_version', 'data_type': 'varchar' }, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
gc_reclaimed	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'process_name', 'data_type': 'varchar'}, {'dimension_name': 'jdk_version', 'data_type': 'varchar' }, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
latency_per_read	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
latency_per_write	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
memory_cached	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
memory_free	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'process_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'jdk_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
memory_used	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
network_bytes_in	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
network_bytes_out	double	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'os_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}, {'dimension_name': 'instance_type', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
task_completed	bigint	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'process_name', 'data_type': 'varchar'}, {'dimension_name': 'jdk_version', 'data_type': 'varchar' }, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}]

measure_name	data_type	Dimensions
task_end_state	varchar	[{'dimension_name': 'availability_zone', 'data_type': 'varchar'}, {'dimension_name': 'microservice_name', 'data_type': 'varchar'}, {'dimension_name': 'instance_name', 'data_type': 'varchar'}, {'dimension_name': 'process_name', 'data_type': 'varchar'}, {'dimension_name': 'jdk_version', 'data_type': 'varchar'}, {'dimension_name': 'cell', 'data_type': 'varchar'}, {'dimension_name': 'region', 'data_type': 'varchar'}, {'dimension_name': 'silo', 'data_type': 'varchar'}]

Example Data

availa ity_z	micro ice_n	insta name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	insta type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9 s ilo-2	r5.4xl e	cpu_l	34:57	0.871		

availa ity_z	micro ice_n	insta name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	insta type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
		silo-2 0000 mazo com												
eu- west- -1	hercu	i- zaZsv k- hercu s- eu- west -1- cell-9 - silo-2 0000 mazo com		AL20		eu- west- - cell-9	eu- west-	eu- west- - cell-9 s ilo-2	r5.4xl e	cpu_i	34:57	3.462		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	cpu_it	34:57	0.102		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	cpu_r	34:57	0.630		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	cpu_s	34:57	0.164		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	cpu_s	34:57	0.107		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	cpu_sm	34:57	0.457		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	cpu_u	34:57	94.20		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	disk_	34:57	72.51		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	disk_ eads	34:57	81.73		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	disk_rites	34:57	77.11		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	disk_	34:57	89.42		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	file_descriptorn_use	34:57	30.08		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom	serve		JDK_	eu-west--cell-9	eu-west-	eu-west--cell-9silo-2		gc_pa	34:57	60.28		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom	serve		JDK_	eu-west--cell-9	eu-west-	eu-west--cell-9silo-2		gc_remed	34:57	75.28		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	latener_re	34:57	8.076		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	latener_wr	34:57	58.11		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	memched	34:57	87.56		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom	serve		JDK_	eu-west--cell-9	eu-west-	eu-west--cell-9silo-2		mem ee	34:57	18.95		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xl e	mem ee	34:57	97.20		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	memed	34:57	12.37		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	netwytes_	34:57	31.02		

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom		AL20		eu-west--cell-9	eu-west-	eu-west--cell-9silo-2	r5.4xle	netwytes_	34:57	0.514		
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom	serve		JDK_	eu-west--cell-9	eu-west-	eu-west--cell-9silo-2		task_leted	34:57		69	

availa ity_z	micro ice_n	instan name	proce ame	os_ve n	jdk_v on	Cell	regio	Silo	instan type	meas ame	Time	meas alue: ble	meas alue: int	measure_v alue::var char
eu-west--1	hercu	i-zaZsvk-hercus-eu-west-1-cell-9-silo-20000mazocom	serve		JDK_	eu-west--cell-9	eu-west-	eu-west--cell-9 s ilo-2		task_state	34:57			SUCCESS_WITH_RESULT

Scheduled query patterns

In this section you will find some common patterns of how you can use Amazon Timestream for LiveAnalytics Scheduled Queries to optimize your dashboards to load faster and at reduced costs. The examples below use a DevOps application scenario to illustrate the key concepts which apply to scheduled queries in general, irrespective of the application scenario.

Scheduled Queries in Timestream for LiveAnalytics allow you to express your queries using the full SQL surface area of Timestream for LiveAnalytics. Your query can include one or more source tables, perform aggregations or any other query allowed by Timestream for LiveAnalytics's SQL language, and then materialize the results of the query in another destination table in Timestream for LiveAnalytics. For ease of exposition, this section refers to this target table of a scheduled query as a *derived table*.

The following are the key points that are covered in this section.

- Using a simple fleet-level aggregate to explain how you can define a scheduled query and understand some basic concepts.

- How you can combine results from the target of a scheduled query (the derived table) with the results from the source table to get the cost and performance benefits of scheduled query.
- What are your trade-offs when configuring the refresh period of the scheduled queries.
- Using scheduled queries for some common scenarios.
  - Tracking the last data point from every instance before a specific date.
  - Distinct values for a dimension to use for populating variables in a dashboard.
- How you handle late arriving data in the context of scheduled queries.
- How you can use one-off manual executions to handle a variety of scenarios not directly covered by automated triggers for scheduled queries.

Topics

- Scenario
- Simple fleet-level aggregates
- Last point from each device
- Unique dimension values
- Handling late-arriving data
- Back-filling historical pre-computations

Scenario

The following examples use a DevOps monitoring scenario which is outlined in Scheduled queries sample schema.

The examples provide the scheduled query definition where you can plug in the appropriate configurations for where to receive execution status notifications for scheduled queries, where to receive reports for errors encountered during execution of a scheduled query, and the IAM role the scheduled query uses to perform its operations.

You can create these scheduled queries after filling in the preceding options, creating the target (or derived) table, and executing the through the AWS CLI. For example, assume that a scheduled query definition is stored in a file, `scheduled_query_example.json`. You can create the query using the CLI command.

```
aws timestream-query create-scheduled-query --cli-input-json  file://
scheduled_query_example.json --profile aws_profile --region us-east-1
```

In the preceding command, the profile passed using the --profile option must have the appropriate permissions to create scheduled queries. See Identity-based policies for Scheduled Queries for detailed instructions for the policies and permissions.

Simple fleet-level aggregates

This first example walks you through some of the basic concepts when working with scheduled queries using a simple example computing fleet-level aggregates. Using this example, you will learn the following.

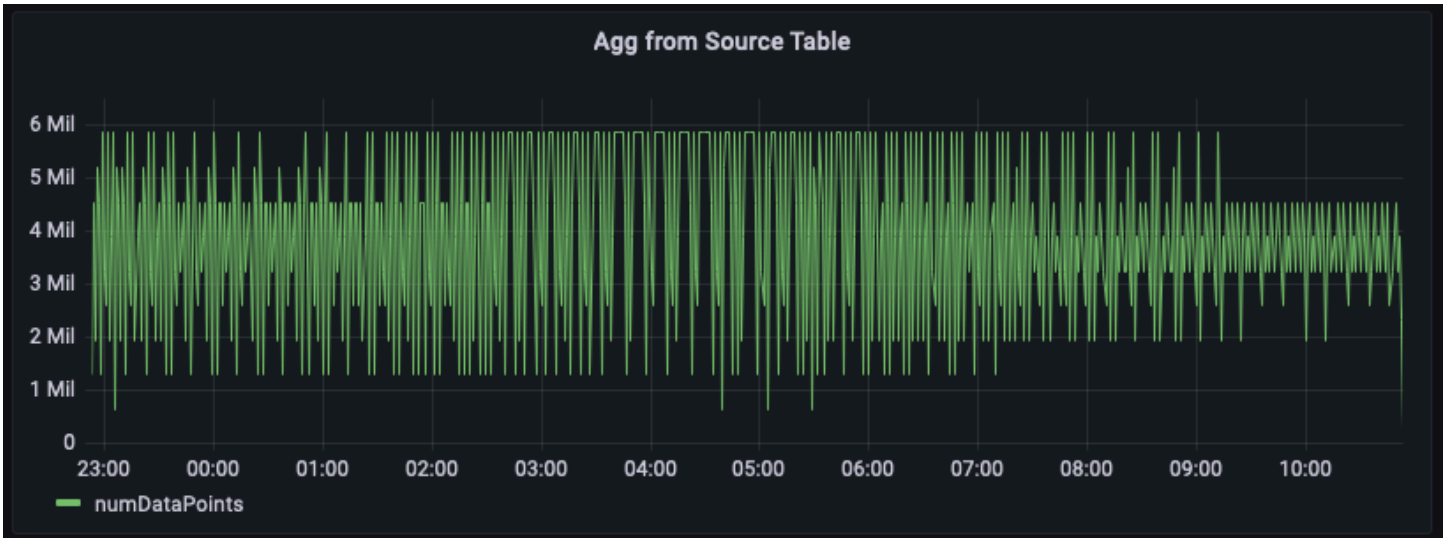
- How to take your dashboard query that is used to obtain aggregate statistics and map it to a scheduled query.
- How Timestream for LiveAnalytics manages the execution of the different instances of your scheduled query.
- How you can have different instances of scheduled queries overlap in time ranges and how the correctness of data is maintained on the target table to ensure that your dashboard using the results of the scheduled query gives you results that match with the same aggregate computed on the raw data.
- How to set the time range and refresh cadence for your scheduled query.
- How you can self-serve track the results of the scheduled queries to tune them so that the execution latency for the query instances are within the acceptable delays of refreshing your dashboards.

Topics

- Aggregate from source tables
- Scheduled query to pre-compute aggregates
- Aggregate from derived table
- Aggregate combining source and derived tables
- Aggregate from frequently refreshed scheduled computation

Aggregate from source tables

In this example, you are tracking the number of metrics emitted by the servers within a given region in every minute. The graph below is an example plotting this time series for the region us-east-1.



Below is an example query to compute this aggregate from the raw data. It filters the rows for the region us-east-1 and then computes the per minute sum by accounting for the 20 metrics (if measure_name is metrics) or 5 events (if measure_name is events). In this example, the graph illustration shows that the number of metrics emitted vary between 1.5 Million to 6 Million per minute. When plotting this time series for several hours (past 12 hours in this figure), this query over the raw data analyzes hundreds of millions of rows.

```
WITH grouped_data AS (
    SELECT region, bin(time, 1m) as minute, SUM(CASE WHEN measure_name = 'metrics' THEN
 20 ELSE 5 END) as numDataPoints
    FROM "raw_data"."devops"
    WHERE time BETWEEN from_milliseconds(1636699996445) AND
 from_milliseconds(1636743196445)
        AND region = 'us-east-1'
    GROUP BY region, measure_name, bin(time, 1m)
)
SELECT minute, SUM(numDataPoints) AS numDataPoints
FROM grouped_data
GROUP BY minute
ORDER BY 1 desc, 2 desc
```

Scheduled query to pre-compute aggregates

If you would like to optimize your dashboards to load faster and lower your costs by scanning less data, you can use a scheduled query to pre-compute these aggregates. Scheduled queries in Timestream for LiveAnalytics allows you to materialize these pre-computations in another Timestream for LiveAnalytics table, which you can subsequently use for your dashboards.

The first step in creating a scheduled query is to identify the query you want to pre-compute. Note that the preceding dashboard was drawn for region us-east-1. However, a different user may want the same aggregate for a different region, say us-west-2 or eu-west-1. To avoid creating a scheduled query for each such query, you can pre-compute the aggregate for each region and materialize the per-region aggregates in another Timestream for LiveAnalytics table.

The query below provides an example of the corresponding pre-computation. As you can see, it is similar to the common table expression grouped_data used in the query on the raw data, except for two differences: 1) it does not use a region predicate, so that we can use one query to pre-compute for all regions; and 2) it uses a parameterized time predicate with a special parameter @scheduled_runtime which is explained in details below.

```
SELECT region, bin(time, 1m) as minute,
    SUM(CASE WHEN measure_name = 'metrics' THEN 20 ELSE 5 END) as numDataPoints
FROM raw_data.devops
WHERE time BETWEEN @scheduled_runtime - 10m AND @scheduled_runtime + 1m
GROUP BY bin(time, 1m), region
```

The preceding query can be converted into a scheduled query using the following specification. The scheduled query is assigned a Name, which is a user-friendly mnemonic. It then includes the QueryString, a ScheduleConfiguration, which is a cron expression. It specifies the TargetConfiguration which maps the query results to the destination table in Timestream for LiveAnalytics. Finally, it specifies a number of other configurations, such as the NotificationConfiguration, where notifications are sent for individual executions of the query, ErrorReportConfiguration where a report is written in case the query encounters any errors, and the ScheduledQueryExecutionRoleArn, which is the role used to perform operations for the scheduled query.

```
{
    "Name": "MultiPT5mPerMinutePerRegionMeasureCount",
    "QueryString": "SELECT region, bin(time, 1m) as minute, SUM(CASE WHEN measure_name
 = 'metrics' THEN 20 ELSE 5 END) as numDataPoints FROM raw_data.devops WHERE time
 BETWEEN @scheduled_runtime - 10m AND @scheduled_runtime + 1m GROUP BY bin(time, 1m),
 region",
    "ScheduleConfiguration": {
        "ScheduleExpression": "cron(0/5 * * * ? *)"
    },
    "NotificationConfiguration": {
        "SnsConfiguration": {
            "TopicArn": "******"
```

```
        }
    },
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName": "derived",
            "TableName": "per_minute_aggs_pt5m",
            "TimeColumn": "minute",
            "DimensionMappings": [
                {
                    "Name": "region",
                    "DimensionValueType": "VARCHAR"
                }
            ],
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "numDataPoints",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "numDataPoints",
                        "MeasureValueType": "BIGINT"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******"
}
```

In the example, the ScheduleExpression cron(0/5 * * * ? *) implies that the query is executed once every 5 minutes at the 5th, 10th, 15th, .. minutes of every hour of every day. These timestamps when a specific instance of this query is triggered is what translates to the @scheduled_runtime parameter used in the query. For instance, consider the instance of this scheduled query executing on 2021-12-01 00:00:00. For this instance, the @scheduled_runtime parameter is initialized to the timestamp 2021-12-01 00:00:00 when invoking the query. Therefore, this specific instance will execute at timestamp 2021-12-01 00:00:00 and will compute the per-minute aggregates from time range 2021-11-30 23:50:00 to 2021-12-01 00:01:00. Similarly, the next instance of this

query is triggered at timestamp 2021-12-01 00:05:00 and in that case, the query will compute per-minute aggregates from the time range 2021-11-30 23:55:00 to 2021-12-01 00:06:00. Hence, the @scheduled_runtime parameter provides a scheduled query to pre-compute the aggregates for the configured time ranges using the invocation time for the queries.

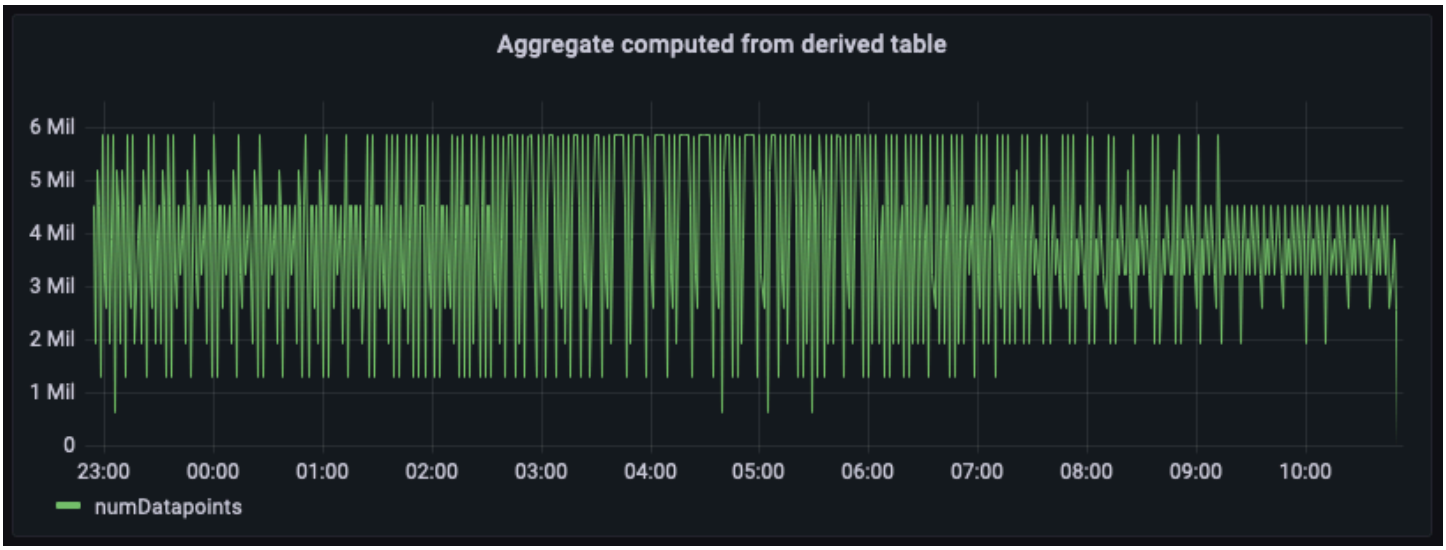
Note that two subsequent instances of the query overlap in their time ranges. This is something you can control based on your requirements. In this case, this overlap allows these queries to update the aggregates based on any data whose arrival was slightly delayed, up to 5 minutes in this example. To ensure correctness of the materialized queries, Timestream for LiveAnalytics ensures that the query at 2021-12-01 00:05:00 will be performed only after the query at 2021-12-01 00:00:00 has completed and the results of the latter queries can update any previously materialized aggregate using if a newer value is generated. For example, if some data at timestamp 2021-11-30 23:59:00 arrived after the query for 2021-12-01 00:00:00 executed but before the query for 2021-12-01 00:05:00, then the execution at 2021-12-01 00:05:00 will recompute the aggregates for the minute 2021-11-30 23:59:00 and this will result in the previous aggregate being updated with the newly-computed value. You can rely on these semantics of the scheduled queries to strike a trade-off between how quickly you update your pre-computations versus how you can gracefully handle some data with delayed arrival. Additional considerations are discussed below on how you trade-off this refresh cadence with freshness of the data and how you address updating the aggregates for data that arrives even more delayed or if your source of the scheduled computation has updated values which would require the aggregates to be recomputed.

Every scheduled computation has a notification configuration where Timestream for LiveAnalytics sends notification of every execution of a scheduled configuration. You can configure an SNS topic for to receive notifications for each invocation. In addition to the success or failure status of a specific instance, it also has several statistics such as the time this computation took to execute, the number of bytes the computation scanned, and the number of bytes the computation wrote to its destination table. You can use these statistics to further tune your query, schedule configuration, or track the spend for your scheduled queries. One aspect worth noting is the execution time for an instance. In this example, the scheduled computation is configured to execute the every 5 minutes. The execution time will determine the delay with which the pre-computation will be available, which will also define the lag in your dashboard when you're using the pre-computed data in your dashboards. Furthermore, if this delay is consistently higher than the refresh interval, for example, if the execution time is more than 5 minutes for a computation configured to refresh every 5 minutes, it is important to tune your computation to run faster to avoid further lag in your dashboards.

Aggregate from derived table

Now that you have set up the scheduled queries and the aggregates are pre-computed and materialized to another Timestream for LiveAnalytics table specified in the target configuration of the scheduled computation, you can use the data in that table to write SQL queries to power your dashboards. Below is an equivalent of the query that uses the materialized pre-aggregates to generate the per minute data point count aggregate for us-east-1.

```
SELECT bin(time, 1m) as minute, SUM(numDataPoints) as numDatapoints
FROM "derived"."per_minute_aggs_pt5m"
WHERE time BETWEEN from_milliseconds(1636699996445) AND
 from_milliseconds(1636743196445)
    AND region = 'us-east-1'
GROUP BY bin(time, 1m)
ORDER BY 1 desc
```



The previous figure plots the aggregate computed from the aggregate table. Comparing this panel with the panel computed from the raw source data, you will notice that they match up exactly, albeit these aggregates are delayed by a few minute, controlled by the refresh interval you configured for the scheduled computation plus the time to execute it.

This query over the pre-computed data scans several orders of magnitude lesser data compared to the aggregates computed over the raw source data. Depending on the granularity of aggregations, this reduction can easily result in 100X lower cost and query latency. There is a cost to executing this scheduled computation. However, depending on how frequently these dashboards are refreshed and how many concurrent users load these dashboards, you end up significantly reducing

your overall costs by using these pre-computations. And this is on top of 10-100X faster load times for the dashboards.

Aggregate combining source and derived tables

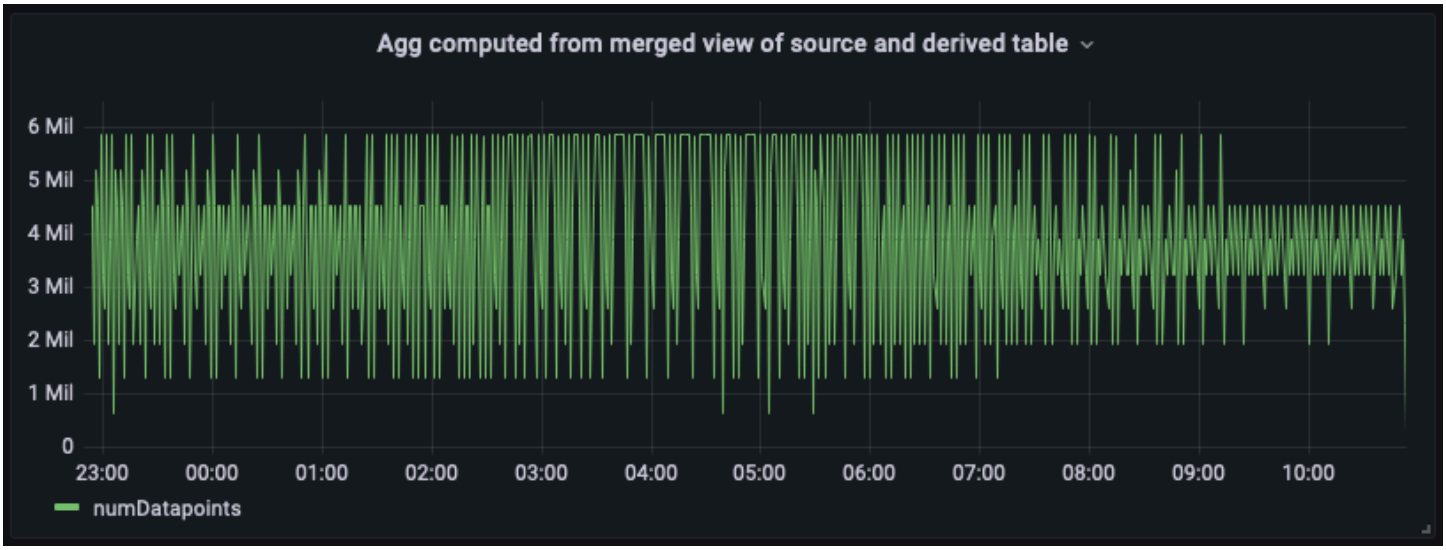
Dashboards created using the derived tables can have a lag. If your application scenario requires the dashboards to have the most recent data, then you can use the power and flexibility of Timestream for LiveAnalytics's SQL support to combine the latest data from the source table with the historical aggregates from the derived table to form a merged view. This merged view uses the union semantics of SQL and non-overlapping time ranges from the source and the derived table. In the example below, we are using the "derived"."per_minute_aggs_pt5m" derived table. Since the scheduled computation for that derived table refreshes once every 5 minutes (per the schedule expression specification), this query below uses the most recent 15 minutes of data from the source table, and any data older than 15 minutes from the derived table and then unions the results to create the merged view that has the best of both worlds: the economics and low latency by reading older pre-computed aggregates from the derived table and the freshness of the aggregates from the source table to power your real time analytics use cases.

Note that this union approach will have slightly higher query latency compared to only querying the derived table and also have slightly higher data scanned, since it is aggregating the raw data in real time to fill in the most recent time interval. However, this merged view will still be significantly faster and cheaper compared to aggregating on the fly from the source table, especially for dashboards rendering days or weeks of data. You can tune the time ranges for this example to suite your application's refresh needs and delay tolerance.

```
WITH aggregated_source_data AS (
    SELECT bin(time, 1m) as minute, SUM(CASE WHEN measure_name = 'metrics' THEN 20 ELSE
 5 END) as numDatapoints
    FROM "raw_data"."devops"
    WHERE time BETWEEN bin(from_milliseconds(1636743196439), 1m) - 15m AND
 from_milliseconds(1636743196439)
        AND region = 'us-east-1'
    GROUP BY bin(time, 1m)
), aggregated_derived_data AS (
    SELECT bin(time, 1m) as minute, SUM(numDataPoints) as numDatapoints
    FROM "derived"."per_minute_aggs_pt5m"
    WHERE time BETWEEN from_milliseconds(1636699996439) AND
 bin(from_milliseconds(1636743196439), 1m) - 15m
        AND region = 'us-east-1'
    GROUP BY bin(time, 1m)
```

```
)
SELECT minute, numDatapoints
FROM (
    (
    SELECT *
    FROM aggregated_derived_data
    )
    UNION
    (
    SELECT *
    FROM aggregated_source_data
    )
)
ORDER BY 1 desc
```

Below is the dashboard panel with this unified merged view. As you can see, the dashboard looks almost identical to the view computed from the derived table, except for that it will have the most up-to-date aggregate at the rightmost tip.



Aggregate from frequently refreshed scheduled computation

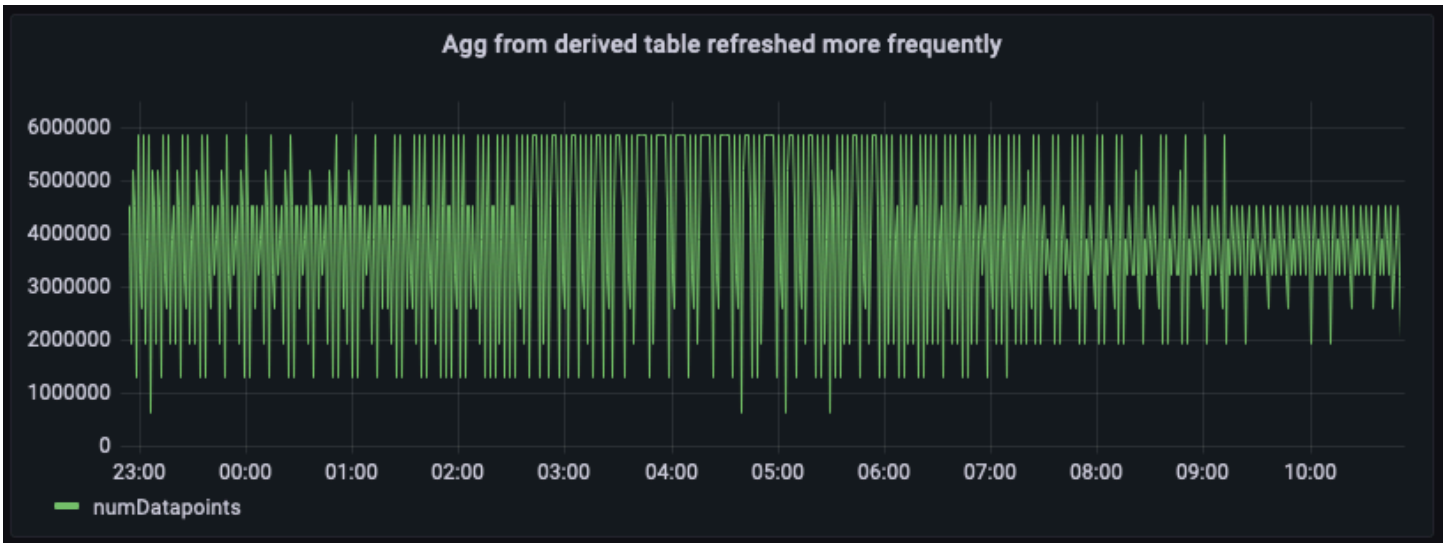
Depending on how frequently your dashboards are loaded and how much latency you want for your dashboard, there is another approach to obtaining fresher results in your dashboard: having the scheduled computation refresh the aggregates more frequently. For instance, below is configuration of the same scheduled computation, except that it refreshes once every minute (note the schedule express cron(0/1 * * * ? *)). With this setup, the derived table per_minute_aggs_pt1m will have much more recent aggregates compared to the scenario where the computation specified a refresh schedule of once every 5 minutes.

```
{
    "Name": "MultiPT1mPerMinutePerRegionMeasureCount",
    "QueryString": "SELECT region, bin(time, 1m) as minute, SUM(CASE WHEN measure_name
 = 'metrics' THEN 20 ELSE 5 END) as numDataPoints FROM raw_data.devops WHERE time
 BETWEEN @scheduled_runtime - 10m AND @scheduled_runtime + 1m GROUP BY bin(time, 1m),
 region",
    "ScheduleConfiguration": {
        "ScheduleExpression": "cron(0/1 * * * ? *)"
    },
    "NotificationConfiguration": {
        "SnsConfiguration": {
            "TopicArn": "******"
        }
    },
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName": "derived",
            "TableName": "per_minute_aggs_pt1m",
            "TimeColumn": "minute",
            "DimensionMappings": [
                {
                    "Name": "region",
                    "DimensionValueType": "VARCHAR"
                }
            ],
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "numDataPoints",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "numDataPoints",
                        "MeasureValueType": "BIGINT"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
```

```
    "ScheduledQueryExecutionRoleArn": "******"
}
```

```
SELECT bin(time, 1m) as minute, SUM(numDataPoints) as numDatapoints
FROM "derived"."per_minute_aggs_pt1m"
WHERE time BETWEEN from_milliseconds(1636699996446) AND
 from_milliseconds(1636743196446)
    AND region = 'us-east-1'
GROUP BY bin(time, 1m), region
ORDER BY 1 desc
```

Since the derived table has more recent aggregates, you can now directly query the derived table per_minute_aggs_pt1m to get fresher aggregates, as can be seen from the previous query and the dashboard snapshot below.



Note that refreshing the scheduled computation at a faster schedule (say 1 minute compared to 5 minutes) will increase the maintenance costs for the scheduled computation. The notification message for every computation's execution provides statistics for how much data was scanned and how much was written to the derived table. Similarly, if you use the merged view to union the derived table, you query costs on the merged view and the dashboard load latency will be higher compared to only querying the derived table. Therefore, the approach you pick will depend on how frequently your dashboards are refreshed and the maintenance costs for the scheduled queries. If you have tens of users refreshing the dashboards once every minute or so, having a more frequent refresh of your derived table will likely result in overall lower costs.

Last point from each device

Your application may require you to read the last measurement emitted by a device. There can be more general use cases to obtain the last measurement for a device before a given date/time or the first measurement for a device after a given date/time. When you have millions of devices and years of data, this search might require scanning large amounts of data.

Below you will see an example of how you can use scheduled queries to optimize searching for the last point emitted by a device. You can use the same pattern to optimize the first point query as well if your application needs them.

Topics

- Computed from source table
- Derived table to precompute at daily granularity
- Computed from derived table
- Combining from source and derived table

Computed from source table

Below is an example query to find the last measurement emitted by the services in a specific deployment (for example, servers for a given micro-service within a given region, cell, silo, and availability_zone). In the example application, this query will return the last measurement for hundreds of servers. Also note that this query has an unbounded time predicate and looks for any data older than a given timestamp.

> **Note**
>
> For information about the `max` and `max_by` functions, see Aggregate functions.

```
SELECT instance_name, MAX(time) AS time, MAX_BY(gc_pause, time) AS last_measure
FROM "raw_data"."devops"
WHERE time < from_milliseconds(1636685271872)
    AND measure_name = 'events'
    AND region = 'us-east-1'
    AND cell = 'us-east-1-cell-10'
    AND silo = 'us-east-1-cell-10-silo-3'
    AND availability_zone = 'us-east-1-1'
    AND microservice_name = 'hercules'
```

```
GROUP BY region, cell, silo, availability_zone, microservice_name,
    instance_name, process_name, jdk_version
ORDER BY instance_name, time DESC
```

Derived table to precompute at daily granularity

You can convert the preceding use case into a scheduled computation. If your application requirements are such that you may need to obtain these values for your entire fleet across multiple regions, cells, silos, availability zones and microservices, you can use one schedule computation to pre-compute the values for your entire fleet. That is the power of Timestream for LiveAnalytics's serverless scheduled queries that allows these queries to scale with your application's scaling requirements.

Below is a query to pre-compute the last point across all the servers for a given day. Note that the query only has a time predicate and not a predicate on the dimensions. The time predicate limits the query to the past day from the time when the computation is triggered based on the specified schedule expression.

```
SELECT region, cell, silo, availability_zone, microservice_name,
    instance_name, process_name, jdk_version,
    MAX(time) AS time, MAX_BY(gc_pause, time) AS last_measure
FROM raw_data.devops
WHERE time BETWEEN bin(@scheduled_runtime, 1d) - 1d AND bin(@scheduled_runtime, 1d)
    AND measure_name = 'events'
GROUP BY region, cell, silo, availability_zone, microservice_name,
    instance_name, process_name, jdk_version
```

Below is a configuration for the scheduled computation using the preceding query which executes that query at 01:00 hrs UTC every day to compute the aggregate for the past day. The schedule expression cron(0 1 * * ? *) controls this behavior and runs an hour after the day has ended to consider any data arriving up to a day late.

```
{
    "Name": "PT1DPerInstanceLastpoint",
    "QueryString": "SELECT region, cell, silo, availability_zone, microservice_name,
 instance_name, process_name, jdk_version, MAX(time) AS time, MAX_BY(gc_pause, time)
 AS last_measure FROM raw_data.devops WHERE time BETWEEN bin(@scheduled_runtime, 1d) -
 1d AND bin(@scheduled_runtime, 1d) AND measure_name = 'events' GROUP BY region, cell,
 silo, availability_zone, microservice_name, instance_name, process_name, jdk_version",
    "ScheduleConfiguration": {
        "ScheduleExpression": "cron(0 1 * * ? *)"
```

```
    },
    "NotificationConfiguration": {
        "SnsConfiguration": {
            "TopicArn": "******"
        }
    },
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName": "derived",
            "TableName": "per_timeseries_lastpoint_pt1d",
            "TimeColumn": "time",
            "DimensionMappings": [
                {
                    "Name": "region",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "cell",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "silo",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "availability_zone",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "microservice_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "instance_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "process_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "jdk_version",
                    "DimensionValueType": "VARCHAR"
                }
```

```
            ],
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "last_measure",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "last_measure",
                        "MeasureValueType": "DOUBLE"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******"
}
```

Computed from derived table

Once you define the derived table using the preceding configuration and at least one instance of the scheduled query has materialized data into the derived table, you can now query the derived table to get the latest measurement. Below is an example query on the derived table.

```
SELECT instance_name, MAX(time) AS time, MAX_BY(last_measure, time) AS last_measure
FROM "derived"."per_timeseries_lastpoint_pt1d"
WHERE time < from_milliseconds(1636746715649)
    AND measure_name = 'last_measure'
    AND region = 'us-east-1'
    AND cell = 'us-east-1-cell-10'
    AND silo = 'us-east-1-cell-10-silo-3'
    AND availability_zone = 'us-east-1-1'
    AND microservice_name = 'hercules'
GROUP BY region, cell, silo, availability_zone, microservice_name,
    instance_name, process_name, jdk_version
ORDER BY instance_name, time DESC
```

Combining from source and derived table

Similar to the previous example, any data from the derived table will not have the most recent writes. Therefore, you can again use a similar pattern as earlier to merge the data from the derived table for the older data and use the source data for the remaining tip. Below is an example of such a query using the similar UNION approach. Since the application requirement is to find the latest measurement before a time period, and this start time can be in past, the way you write this query is to use the provided time, use the source data for up to a day old from the specified time, and then use the derived table on the older data. As you can see from the query example below, the time predicate on the source data is bounded. That ensures efficient processing on the source table which has significantly higher volume of data, and then the unbounded time predicate is on the derived table.

```
WITH last_point_derived AS (
    SELECT instance_name, MAX(time) AS time, MAX_BY(last_measure, time) AS last_measure
    FROM "derived"."per_timeseries_lastpoint_pt1d"
    WHERE time < from_milliseconds(1636746715649)
        AND measure_name = 'last_measure'
        AND region = 'us-east-1'
        AND cell = 'us-east-1-cell-10'
        AND silo = 'us-east-1-cell-10-silo-3'
        AND availability_zone = 'us-east-1-1'
        AND microservice_name = 'hercules'
    GROUP BY region, cell, silo, availability_zone, microservice_name,
        instance_name, process_name, jdk_version
), last_point_source AS (
    SELECT instance_name, MAX(time) AS time, MAX_BY(gc_pause, time) AS last_measure
    FROM "raw_data"."devops"
    WHERE time < from_milliseconds(1636746715649) AND time >
 from_milliseconds(1636746715649) - 26h
        AND measure_name = 'events'
        AND region = 'us-east-1'
        AND cell = 'us-east-1-cell-10'
        AND silo = 'us-east-1-cell-10-silo-3'
        AND availability_zone = 'us-east-1-1'
        AND microservice_name = 'hercules'
    GROUP BY region, cell, silo, availability_zone, microservice_name,
        instance_name, process_name, jdk_version
)
SELECT instance_name, MAX(time) AS time, MAX_BY(last_measure, time) AS last_measure
FROM (
    SELECT * FROM last_point_derived
```

```
    UNION
    SELECT * FROM last_point_source
)
GROUP BY instance_name
ORDER BY instance_name, time DESC
```

The previous is just one illustration of how you can structure the derived tables. If you have years of data, you can use more levels of aggregations. For instance, you can have monthly aggregates on top of daily aggregates, and you can have hourly aggregates before the daily. So you can merge together the most recent to fill in the last hour, the hourly to fill in the last day, the daily to fill in the last month, and monthly to fill in the older. The number of levels you set up vs. the refresh schedule will be depending on your requirements of how frequently these queries are issues and how many users are concurrently issuing these queries.

Unique dimension values

You may have a use case where you have dashboards which you want to use the unique values of dimensions as variables to drill down on the metrics corresponding to a specific slice of data. The snapshot below is an example where the dashboard pre-populates the unique values of several dimensions such as region, cell, silo, microservice, and availability_zone. Here we show an example of how you can use scheduled queries to significantly speed up computing these distinct values of these variables from the metrics you are tracking.

Topics

- On raw data
- Pre-compute unique dimension values
- Computing the variables from derived table

On raw data

You can use SELECT DISTINCT to compute the distinct values seen from your data. For instance, if you want to obtain the distinct values of region, you can use the query of this form.

```
SELECT DISTINCT region
FROM "raw_data"."devops"
WHERE time > ago(1h)
ORDER BY 1
```

You may be tracking millions of devices and billions of time series. However, in most cases, these interesting variables are for lower cardinality dimensions, where you have a few to tens of values. Computing DISTINCT from raw data can require scanning large volumes of data.

Pre-compute unique dimension values

You want these variables to load fast so that your dashboards are interactive. Moreover, these variables are often computed on every dashboard load, so you want them to be cost-effective as well. You can optimize finding these variables using scheduled queries and materializing them in a derived table.

First, you need to identify the dimensions for which you need to compute the DISTINCT values or columns which you will use in the predicates when computing the DISTINCT value.

In this example, you can see that the dashboard is populating distinct values for the dimensions region, cell, silo, availability_zone and microservice. So you can use the query below to pre-compute these unique values.

```
SELECT region, cell, silo, availability_zone, microservice_name,
    min(@scheduled_runtime) AS time, COUNT(*) as numDataPoints
FROM raw_data.devops
WHERE time BETWEEN @scheduled_runtime - 15m AND @scheduled_runtime
GROUP BY region, cell, silo, availability_zone, microservice_name
```

There are a few important things to note here.

- You can use one scheduled computation to pre-compute values for many different queries. For instance, you are using the preceding query to pre-compute values for five different variables. So you don't need one for each variable. You can use this same pattern to identify shared computation across multiple panels to optimize the number of scheduled queries you need to maintain.
- The unique values of the dimensions isn't inherently time series data. So you convert this to time series using the @scheduled_runtime. By associating this data with the @scheduled_runtime parameter, you can also track which unique values appeared at a given point in time, thus creating time series data out of it.
- In the previous example, you will see a metric value being tracked. This example uses COUNT(*). You can compute other meaningful aggregates if you want to track them for your dashboards.

Below is a configuration for a scheduled computation using the previous query. In this example, it is configured to refresh once every 15 mins using the schedule expression cron(0/15 * * * ? *).

```
{
    "Name": "PT15mHighCardPerUniqueDimensions",
    "QueryString": "SELECT region, cell, silo, availability_zone, microservice_name,
 min(@scheduled_runtime) AS time, COUNT(*) as numDataPoints FROM raw_data.devops WHERE
 time BETWEEN @scheduled_runtime - 15m AND @scheduled_runtime GROUP BY region, cell,
 silo, availability_zone, microservice_name",
    "ScheduleConfiguration": {
        "ScheduleExpression": "cron(0/15 * * * ? *)"
    },
    "NotificationConfiguration": {
        "SnsConfiguration": {
            "TopicArn": "******"
        }
    },
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName": "derived",
            "TableName": "hc_unique_dimensions_pt15m",
            "TimeColumn": "time",
            "DimensionMappings": [
                {
                    "Name": "region",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "cell",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "silo",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "availability_zone",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "microservice_name",
                    "DimensionValueType": "VARCHAR"
                }
```

```
            ],
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "count_multi",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "numDataPoints",
                        "MeasureValueType": "BIGINT"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******"
}
```

Computing the variables from derived table

Once the scheduled computation pre-materializes the unique values in the derived table hc_unique_dimensions_pt15m, you can use the derived table to efficiently compute the unique values of the dimensions. Below are example queries for how to compute the unique values, and how you can use other variables as predicates in these unique value queries.

Region

```
SELECT DISTINCT region
FROM "derived"."hc_unique_dimensions_pt15m"
WHERE time > ago(1h)
ORDER BY 1
```

Cell

```
SELECT DISTINCT cell
FROM "derived"."hc_unique_dimensions_pt15m"
WHERE time > ago(1h)
  AND region = '${region}'
```

```
ORDER BY 1
```

Silo

```
SELECT DISTINCT silo
FROM "derived"."hc_unique_dimensions_pt15m"
WHERE time > ago(1h)
   AND region = '${region}' AND cell = '${cell}'
ORDER BY 1
```

Microservice

```
SELECT DISTINCT microservice_name
FROM "derived"."hc_unique_dimensions_pt15m"
WHERE time > ago(1h)
   AND region = '${region}' AND cell = '${cell}'
ORDER BY 1
```

Availability Zone

```
SELECT DISTINCT availability_zone
FROM "derived"."hc_unique_dimensions_pt15m"
WHERE time > ago(1h)
   AND region = '${region}' AND cell = '${cell}' AND silo = '${silo}'
ORDER BY 1
```

Handling late-arriving data

You may have scenarios where you can have data that arrives significantly late, for example, the time when the data was ingested into Timestream for LiveAnalytics is significantly delayed compared to the timestamp associated to the rows that are ingested. In the previous examples, you have seen how you can use the time ranges defined by the @scheduled_runtime parameter to account for some late arriving data. However, if you have use cases where data can be delayed by hours or days, you may need a different pattern to make sure your pre-computations in the derived table are appropriately updated to reflect such late-arriving data. For general information about late-arriving data, see Writing data (inserts and upserts).

In the following you will see two different ways to address this late arriving data.

- If you have predictable delays in your data arrival, then you can use another "catch-up" scheduled computation to update your aggregates for late arriving data.

- If you have un-predictable delays or occasional late-arrival data, you can use manual executions to update the derived tables.

This discussion covers scenarios for late data arrival. However, the same principles apply for data corrections, where you have modified the data in your source table and you want to update the aggregates in your derived tables.

Topics

- Scheduled catch-up queries
- Manual executions for unpredictable late arriving data

Scheduled catch-up queries

Query aggregating data that arrived in time

Below is a pattern you will see how you can use an automated way to update your aggregates if you have predictable delays in your data arrival. Consider one of the previous examples of a scheduled computation on real-time data below. This scheduled computation refreshes the derived table once every 30 minutes and already accounts for data up to an hour delayed.

```
{
    "Name": "MultiPT30mPerHrPerTimeseriesDPCount",
    "QueryString": "SELECT region, cell, silo, availability_zone, microservice_name,
 instance_type, os_version, instance_name, process_name, jdk_version, bin(time,
 1h) as hour, SUM(CASE WHEN measure_name = 'metrics' THEN 20 ELSE 5 END) as
 numDataPoints FROM raw_data.devops WHERE time BETWEEN bin(@scheduled_runtime, 1h)
 - 1h AND @scheduled_runtime + 1h GROUP BY region, cell, silo, availability_zone,
 microservice_name, instance_type, os_version, instance_name, process_name,
 jdk_version, bin(time, 1h)",
    "ScheduleConfiguration": {
        "ScheduleExpression": "cron(0/30 * * * ? *)"
    },
    "NotificationConfiguration": {
        "SnsConfiguration": {
            "TopicArn": "******"
        }
    },
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName": "derived",
```

```
            "TableName": "dp_per_timeseries_per_hr",
            "TimeColumn": "hour",
            "DimensionMappings": [
                {
                    "Name": "region",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "cell",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "silo",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "availability_zone",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "microservice_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "instance_type",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "os_version",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "instance_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "process_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "jdk_version",
                    "DimensionValueType": "VARCHAR"
                }
            ],
```

```
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "numDataPoints",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "numDataPoints",
                        "MeasureValueType": "BIGINT"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******"
}
```

Catch-up query updating the aggregates for late arriving data

Now if you consider the case that your data can be delayed by about 12 hours. Below is a variant of the same query. However, the difference is that it computes the aggregates on data that is delayed by up to 12 hours compared to when the scheduled computation is being triggered. For instance, you see the query in the example below, the time range this query is targeting is between 2h to 14h before when the query is triggered. Moreover, if you notice the schedule expression cron(0 0,12 * * ? *), it will trigger the computation at 00:00 UTC and 12:00 UTC every day. Therefore, when the query is triggered on 2021-12-01 00:00:00, then the query updates aggregates in the time range 2021-11-30 10:00:00 to 2021-11-30 22:00:00. Scheduled queries use upsert semantics similar to Timestream for LiveAnalytics's writes where this catch-up query will update the aggregate values with newer values if there is late arriving data in the window or if newer aggregates are found (e.g., a new grouping shows up in this aggregate which was not present when the original scheduled computation was triggered), then the new aggregate will be inserted into the derived table. Similarly, when the next instance is triggered on 2021-12-01 12:00:00, then that instance will update aggregates in the range 2021-11-30 22:00:00 to 2021-12-01 10:00:00.

```
        {
    "Name": "MultiPT12HPerHrPerTimeseriesDPCountCatchUp",
```

```
   "QueryString": "SELECT region, cell, silo, availability_zone, microservice_name,
 instance_type, os_version, instance_name, process_name, jdk_version, bin(time, 1h)
 as hour, SUM(CASE WHEN measure_name = 'metrics' THEN 20 ELSE 5 END) as numDataPoints
 FROM raw_data.devops WHERE time BETWEEN bin(@scheduled_runtime, 1h) - 14h AND
 bin(@scheduled_runtime, 1h) - 2h GROUP BY region, cell, silo, availability_zone,
 microservice_name, instance_type, os_version, instance_name, process_name,
 jdk_version, bin(time, 1h)",
   "ScheduleConfiguration": {
       "ScheduleExpression": "cron(0 0,12 * * ? *)"
   },
   "NotificationConfiguration": {
       "SnsConfiguration": {
           "TopicArn": "******"
       }
   },
   "TargetConfiguration": {
       "TimestreamConfiguration": {
           "DatabaseName": "derived",
           "TableName": "dp_per_timeseries_per_hr",
           "TimeColumn": "hour",
           "DimensionMappings": [
               {
                   "Name": "region",
                   "DimensionValueType": "VARCHAR"
               },
               {
                   "Name": "cell",
                   "DimensionValueType": "VARCHAR"
               },
               {
                   "Name": "silo",
                   "DimensionValueType": "VARCHAR"
               },
               {
                   "Name": "availability_zone",
                   "DimensionValueType": "VARCHAR"
               },
               {
                   "Name": "microservice_name",
                   "DimensionValueType": "VARCHAR"
               },
               {
                   "Name": "instance_type",
                   "DimensionValueType": "VARCHAR"
```

```
                },
                {
                    "Name": "os_version",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "instance_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "process_name",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "jdk_version",
                    "DimensionValueType": "VARCHAR"
                }
            ],
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "numDataPoints",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "numDataPoints",
                        "MeasureValueType": "BIGINT"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******"
}
```

This preceding example is an illustration assuming your late arrival is bounded to 12 hours and it is okay to update the derived table once every 12 hours for data arriving later than the real time window. You can adapt this pattern to update your derived table once every hour so your derived

table reflects the late arriving data sooner. Similarly, you can adapt the time range to be older than 12 hours, e.g., a day or even a week or more, to handle predictable late-arriving data.

Manual executions for unpredictable late arriving data

There can be instances where you have unpredictable late arriving data or you made changes to the source data and updated some values after the fact. In all such cases, you can manually trigger scheduled queries to update the derived table. Below is an example on how you can achieve this.

Assume that you have the use case where you have the computation written to the derived table dp_per_timeseries_per_hr. Your base data in the table devops was updated in the time range 2021-11-30 23:00:00 - 2021-12-01 00:00:00. There are two different scheduled queries that can be used to update this derived table: MultiPT30mPerHrPerTimeseriesDPCount and MultiPT12HPerHrPerTimeseriesDPCountCatchUp. Each scheduled computation you create in Timestream for LiveAnalytics has a unique ARN which you obtain when you create the computation or when you perform a list operation. You can use the ARN for the computation and a value for the parameter @scheduled_runtime taken by the query to perform this operation.

Assume that the computation for MultiPT30mPerHrPerTimeseriesDPCount has an ARN arn_1 and you want to use this computation to update the derived table. Since the preceding scheduled computation updates the aggregates 1h before and 1hr after the @scheduled_runtime value, you can cover the time range for the update (2021-11-30 23:00:00 - 2021-12-01 00:00:00) using a value of 2021-12-01 00:00:00 for the @scheduled_runtime parameter. You can use the ExecuteScheduledQuery API to pass the ARN of this computation and the time parameter value in epoch seconds (in UTC) to achieve this. Below is an example using the AWS CLI and you can follow the same pattern using any of the SDKs supported by Timestream for LiveAnalytics.

```
aws timestream-query execute-scheduled-query --scheduled-query-arn arn_1 --invocation-
time 1638316800 --profile profile --region us-east-1
```

In the previous example, profile is the AWS profile which has the appropriate privileges to make this API call and 1638316800 corresponds to the epoch second for 2021-12-01 00:00:00. This manual trigger behaves almost like the automated trigger assuming the system triggered this invocation at the desired time period.

If you had an update in a longer time period, say the base data was updated for 2021-11-30 23:00:00 - 2021-12-01 11:00:00, then you can trigger the preceding queries multiple times to cover this entire time range. For instance, you could do six different execution as follows.

```
aws timestream-query execute-scheduled-query --scheduled-query-arn arn_1 --invocation-
time 1638316800 --profile profile --region us-east-1

aws timestream-query execute-scheduled-query --scheduled-query-arn arn_1 --invocation-
time 1638324000 --profile profile --region us-east-1

aws timestream-query execute-scheduled-query --scheduled-query-arn arn_1 --invocation-
time 1638331200 --profile profile --region us-east-1

aws timestream-query execute-scheduled-query --scheduled-query-arn arn_1 --invocation-
time 1638338400 --profile profile --region us-east-1

aws timestream-query execute-scheduled-query --scheduled-query-arn arn_1 --invocation-
time 1638345600 --profile profile --region us-east-1

aws timestream-query execute-scheduled-query --scheduled-query-arn arn_1 --invocation-
time 1638352800 --profile profile --region us-east-1
```

The previous six commands correspond to the scheduled computation invoked at 2021-12-01 00:00:00, 2021-12-01 02:00:00, 2021-12-01 04:0:00, 2021-12-01 06:00:00, 2021-12-01 08:00:00, and 2021-12-01 10:00:

Alternatively, you can use the computation MultiPT12HPerHrPerTimeseriesDPCountCatchUp triggered at 2021-12-01 13:00:00 for one execution to update the aggregates for the entire 12 hour time range. For instance, if arn_2 is the ARN for that computation, you can execute the following command from CLI.

```
aws timestream-query execute-scheduled-query --scheduled-query-arn arn_2 --invocation-
time 1638363600 --profile profile --region us-east-1
```

It is worth noting that for a manual trigger, you can use a timestamp for the invocation-time parameter that does not need to be aligned with that automated trigger timestamps. For instance, in the previous example, you triggered the computation at time 2021-12-01 13:00:00 even though the automated schedule only triggers at timestamps 2021-12-01 10:00:00, 2021-12-01 12:00:00, and 2021-12-02 00:00:00. Timestream for LiveAnalytics provides you with the flexibility to trigger it with appropriate values as needed for your manual operations.

Following are a few important considerations when using the ExecuteScheduledQuery API.

- If you are triggering multiple of these invocations, you need to make sure that these invocations do not generate results in overlapping time ranges. For instance, in the previous examples, there

were six invocations. Each invocation covers 2 hours of time range, and hence the invocation timestamps were spread out by two hours each to avoid any overlap in the updates. This ensures that the data in the derived table ends up in a state that matches are aggregates from the source table. If you cannot ensure non-overlapping time ranges, then make sure these the executions are triggered sequentially one after the other. If you trigger multiple executions concurrently which overlap in their time ranges, then you can see trigger failures where you might see version conflicts in the error reports for these executions. Results generated by a scheduled query invocation are assigned a version based on when the invocation was triggered. Therefore, rows generated by newer invocations have higher versions. A higher version record can overwrite a lower version record. For automatically-triggered scheduled queries, Timestream for LiveAnalytics automatically manages the schedules so that you don't see these issues even if the subsequent invocations have overlapping time ranges.

- noted earlier, you can trigger the invocations with any timestamp value for @scheduled_runtime. So it is your responsibility to appropriately set the values so the appropriate time ranges are updated in the derived table corresponding to the ranges where data was updated in the source table.
- You can also use these manual trigger for scheduled queries that are in the DISABLED state. This allows you to define special queries that are not executed in an automated schedule, since they are in the DISABLED state. Rather, you can use the manual triggers on them to manage data corrections or late arrival use cases.

Back-filling historical pre-computations

When you create a scheduled computation, Timestream for LiveAnalytics manages executions of the queries moving forward where the refresh is governed by the schedule expression you provide. Depending of how much historical data your source table, you may want to update your derived table with aggregates corresponding to the historical data. You can use the preceding logic for manual triggers to back-fill the historical aggregates.

For instance, if we consider the derived table per_timeseries_lastpoint_pt1d, then the scheduled computation is updated once a day for the past day. If your source table has a year of data, you can use the ARN for this scheduled computation and trigger it manually for every day up to a year old so that the derived table has all the historical queries populated. Notes that all the caveats for manual triggers apply here. Moreover, if the derived table is set up in a way that the historical ingestion will write to magnetic store on the derived table, be aware of the best practices and limits for writes to the magnetic store.

Scheduled query examples

This section contains examples of how you can use Timestream for LiveAnalytics's Scheduled Queries to optimize the costs and dashboard load times when visualizing fleet-wide statistics effectively monitor your fleet of devices. Scheduled Queries in Timestream for LiveAnalytics allow you to express your queries using the full SQL surface area of Timestream for LiveAnalytics. Your query can include one or more source tables, perform aggregations or any other query allowed by Timestream for LiveAnalytics's SQL language, and then store the results of the query in another destination table in Timestream for LiveAnalytics.

This section refers to the target table of a scheduled query as a *derived table*.

As an example, we will use a DevOps application where you are monitoring a large fleet of servers that are deployed across multiple deployments (such as regions, cells, and silos), multiple microservices, and you're tracking the fleet-wide statistics using Timestream for LiveAnalytics. The example schema we will use is described in Scheduled Queries Sample Schema.

The following scenarios will be described.

- How to convert a dashboard, plotting aggregated statistics from the raw data you ingest into Timestream for LiveAnalytics into a scheduled query and then how to use your pre-computed aggregates to create a new dashboard showing aggregate statistics.
- How to combine scheduled queries to get an aggregate view and the raw granular data, to drill down into details. This allows you to store and analyze the raw data while optimizing your common fleet-wide operations using scheduled queries.
- How to optimize costs using scheduled queries by finding which aggregates are used in multiple dashboards and have the same scheduled query populate multiple panels in the same or multiple dashboards.

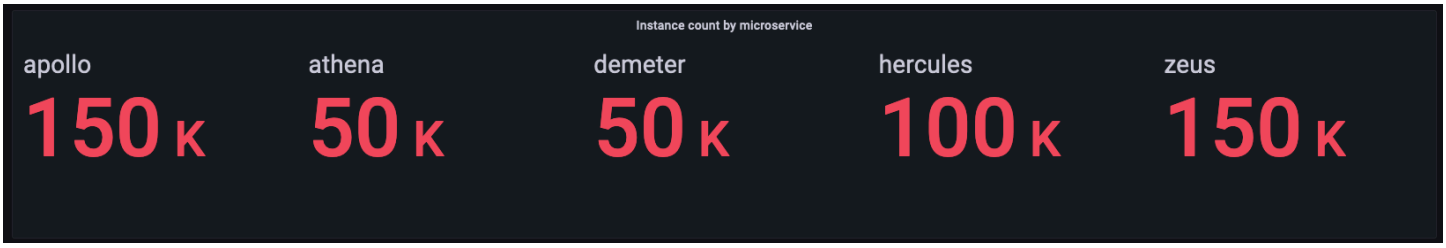
Topics

- Converting an aggregate dashboard to scheduled query
- Using scheduled queries and raw data for drill downs
- Optimizing costs by sharing scheduled query across dashboards
- Comparing a query on a base table with a query of scheduled query results

Converting an aggregate dashboard to scheduled query

Assume you are computing the fleet-wide statistics such as host counts in the fleet by the five microservices and by the six regions where your service is deployed. From the snapshot below, you can see there are 500K servers emitting metrics, and some of the bigger regions (e.g., us-east-1) have >200K servers.

Computing these aggregates, where you are computing distinct instance names over hundreds of gigabytes of data can result in query latency of tens of seconds, in addition to the cost of scanning the data.



Original dashboard query

The aggregate shown in the dasboard panel is computed, from raw data, using the query below. The query uses multiple SQL constructs, such as distinct counts and multiple aggregation functions.

```
SELECT CASE WHEN microservice_name = 'apollo' THEN num_instances ELSE NULL END AS
 apollo,
    CASE WHEN microservice_name = 'athena' THEN num_instances ELSE NULL END AS athena,
    CASE WHEN microservice_name = 'demeter' THEN num_instances ELSE NULL END AS
 demeter,
    CASE WHEN microservice_name = 'hercules' THEN num_instances ELSE NULL END AS
 hercules,
    CASE WHEN microservice_name = 'zeus' THEN num_instances ELSE NULL END AS zeus
FROM (
    SELECT microservice_name, SUM(num_instances) AS num_instances
    FROM (
        SELECT microservice_name, COUNT(DISTINCT instance_name) as num_instances
        FROM "raw_data"."devops"
        WHERE time BETWEEN from_milliseconds(1636526171043) AND
 from_milliseconds(1636612571043)
            AND measure_name = 'metrics'
        GROUP BY region, cell, silo, availability_zone, microservice_name
    )
    GROUP BY microservice_name
```

```
)
```

Converting to a scheduled query

The previous query can be converted into a scheduled query as follows. You first compute the distinct host names within a given deployment in a region, cell, silo, availability zone and microservice. Then you add up the hosts to compute a per hour per microservice host count. By using the `@scheduled_runtime` parameter supported by the scheduled queries, you can recompute it for the past hour when the query is invoked. The `bin(@scheduled_runtime, 1h)` in the `WHERE` clause of the inner query ensures that even if the query is scheduled at a time in the middle of the hour, you still get the data for the full hour.

Even though the query computes hourly aggregates, as you will see in the scheduled computation configuration, it is set up to refresh every half hour so that you get updates in your derived table sooner. You can tune that based on your freshness requirements, e.g., recompute the aggregates every 15 minutes or recompute it at the hour boundaries.

```
SELECT microservice_name, hour, SUM(num_instances) AS num_instances
FROM (
        SELECT microservice_name, bin(time, 1h) AS hour,
            COUNT(DISTINCT instance_name) as num_instances
       FROM raw_data.devops
       WHERE time BETWEEN bin(@scheduled_runtime, 1h) - 1h AND @scheduled_runtime

           AND measure_name = 'metrics'
       GROUP BY region, cell, silo, availability_zone, microservice_name, bin(time, 1h)

     )
GROUP BY microservice_name, hour
```

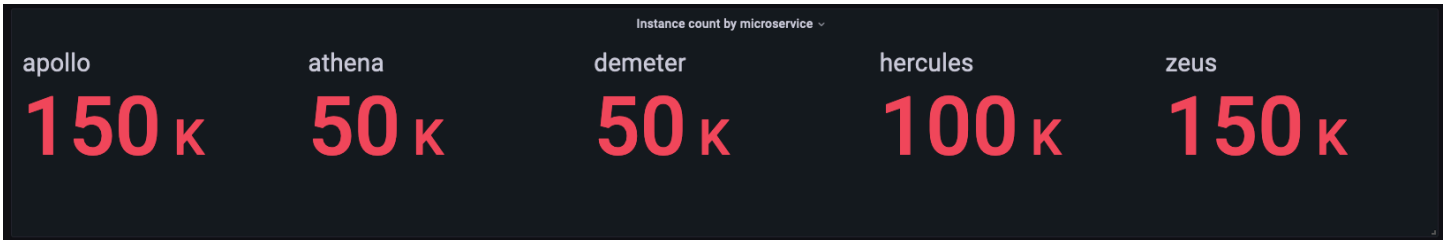
```
{
    "Name": "MultiPT30mHostCountMicroservicePerHr",
    "QueryString": "SELECT microservice_name, hour, SUM(num_instances) AS num_instances
    FROM (        SELECT microservice_name, bin(time, 1h) AS hour, COUNT(DISTINCT
 instance_name) as num_instances        FROM raw_data.devops        WHERE time BETWEEN
 bin(@scheduled_runtime, 1h) - 1h AND @scheduled_runtime            AND measure_name
 = 'metrics'        GROUP BY region, cell, silo, availability_zone, microservice_name,
 bin(time, 1h)    )    GROUP BY microservice_name, hour",
    "ScheduleConfiguration": {
        "ScheduleExpression": "cron(0/30 * * * ? *)"
    },
```

```
    "NotificationConfiguration": {
        "SnsConfiguration": {
            "TopicArn": "******"
        }
    },
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName": "derived",
            "TableName": "host_count_pt1h",
            "TimeColumn": "hour",
            "DimensionMappings": [
                {
                    "Name": "microservice_name",
                    "DimensionValueType": "VARCHAR"
                }
            ],
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "num_instances",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "num_instances",
                        "MeasureValueType": "BIGINT"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******"
}
```

Using the pre-computed results in a new dashboard

You will now see how to create your aggregate view dashboard using the derived table from the scheduled query you created. From the dashboard snapshot, you will also be able to validate that the aggregates computed from the derived table and the base table also match. Once you create the dashboards using the derived tables, you will notice the significantly faster load time and lower

costs of using the derived tables compared to computing these aggregates from the raw data. Below is a snapshot of the dashboard using pre-computed data, and the query used to render this panel using pre-computed data stored in the table "derived"."host_count_pt1h". Note that the structure of the query is very similar to the query that was used in the dashboard on raw data, except that is it using the derived table which already computes the distinct counts which this query is aggregating.




```
SELECT CASE WHEN microservice_name = 'apollo' THEN num_instances ELSE NULL END AS
 apollo,
    CASE WHEN microservice_name = 'athena' THEN num_instances ELSE NULL END AS athena,
    CASE WHEN microservice_name = 'demeter' THEN num_instances ELSE NULL END AS
 demeter,
    CASE WHEN microservice_name = 'hercules' THEN num_instances ELSE NULL END AS
 hercules,
    CASE WHEN microservice_name = 'zeus' THEN num_instances ELSE NULL END AS zeus
FROM (
    SELECT microservice_name, AVG(num_instances) AS num_instances
    FROM (
        SELECT microservice_name, bin(time, 1h), SUM(num_instances) as num_instances
        FROM "derived"."host_count_pt1h"
        WHERE time BETWEEN from_milliseconds(1636567785421) AND
 from_milliseconds(1636654185421)
            AND measure_name = 'num_instances'
        GROUP BY microservice_name, bin(time, 1h)
    )
    GROUP BY microservice_name
)
```

Using scheduled queries and raw data for drill downs

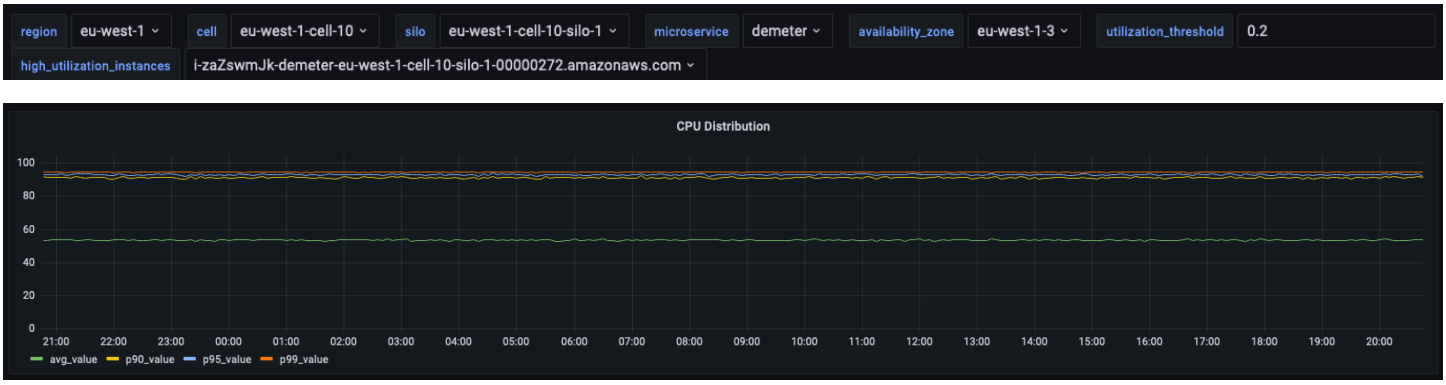
You can use the aggregated statistics across your fleet to identify areas that need drill downs and then use the raw data to drill down into granular data to get deeper insights.

In this example, you will see how you can use aggregate dashboard to identify any deployment (a deployment is for a given microservice within a given region, cell, silo, and availability zone) which

seems to have higher CPU utilization compared to other deployments. You can then drill down to get a better understanding using the raw data. Since these drill downs might be infrequent and only access data relevant to the deployment, you can use the raw data for this analysis and do not need to use scheduled queries.

Per deployment drill down

The dashboard below provides drill down into more granular and server-level statistics within a given deployment. To help you drill down into the different parts of your fleet, this dashboard uses variables such as region, cell, silo, microservice, and availability_zone. It then shows some aggregate statistics for that deployment.



In the query below, you can see that the values chosen in the drop down of the variables are used as predicates in the `WHERE` clause of the query, which allows you to only focus on the data for the deployment. And then the panel plots the aggregated CPU metrics for instances in that deployment. You can use the raw data to perform this drill down with interactive query latency to derive deeper insights.

```
SELECT bin(time, 5m) as minute,
    ROUND(AVG(cpu_user), 2) AS avg_value,
    ROUND(APPROX_PERCENTILE(cpu_user, 0.9), 2) AS p90_value,
    ROUND(APPROX_PERCENTILE(cpu_user, 0.95), 2) AS p95_value,
    ROUND(APPROX_PERCENTILE(cpu_user, 0.99), 2) AS p99_value
FROM "raw_data"."devops"
WHERE time BETWEEN from_milliseconds(1636527099476) AND
 from_milliseconds(1636613499476)
    AND region = 'eu-west-1'
    AND cell = 'eu-west-1-cell-10'
    AND silo = 'eu-west-1-cell-10-silo-1'
    AND microservice_name = 'demeter'
    AND availability_zone = 'eu-west-1-3'
    AND measure_name = 'metrics'
```

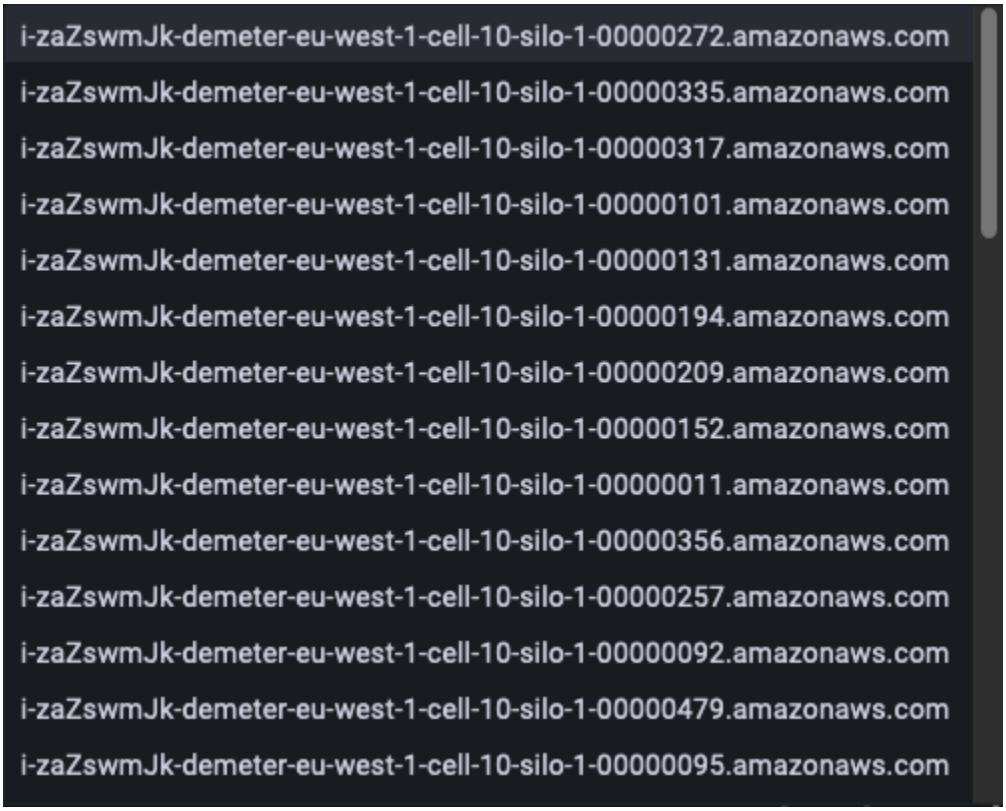
```
GROUP BY bin(time, 5m)
ORDER BY 1
```

Instance-level statistics

This dashboard further computes another variable that also lists the servers/instances with high CPU utilization, sorted in descending order of utilization. The query used to compute this variable is displayed below.

```
WITH microservice_cell_avg AS (
    SELECT AVG(cpu_user) AS microservice_avg_metric
    FROM "raw_data"."devops"
    WHERE $__timeFilter
        AND measure_name = 'metrics'
        AND region = '${region}'
        AND cell = '${cell}'
        AND silo = '${silo}'
        AND availability_zone = '${availability_zone}'
        AND microservice_name = '${microservice}'
), instance_avg AS (
    SELECT instance_name,
        AVG(cpu_user) AS instance_avg_metric
    FROM "raw_data"."devops"
    WHERE $__timeFilter
        AND measure_name = 'metrics'
        AND region = '${region}'
        AND cell = '${cell}'
        AND silo = '${silo}'
        AND microservice_name = '${microservice}'
        AND availability_zone = '${availability_zone}'
    GROUP BY availability_zone, instance_name
)
SELECT i.instance_name
FROM instance_avg i CROSS JOIN microservice_cell_avg m
WHERE i.instance_avg_metric > (1 + ${utilization_threshold}) *
 m.microservice_avg_metric
ORDER BY i.instance_avg_metric DESC
```

In the preceding query, the variable is dynamically recalculated depending on the values chosen for the other variables. Once the variable is populated for a deployment, you can pick individual instances from the list to further visualize the metrics from that instance. You can pick the different instances from the drop down of the instance names as seen from the snapshot below.



Preceding panels show the statistics for the instance that is selected and below are the queries used to fetch these statistics.

```
SELECT BIN(time, 30m) AS time_bin,
    AVG(cpu_user) AS avg_cpu,
    ROUND(APPROX_PERCENTILE(cpu_user, 0.99), 2) as p99_cpu
```

```
FROM "raw_data"."devops"
WHERE time BETWEEN from_milliseconds(1636527099477) AND
 from_milliseconds(1636613499477)
    AND measure_name = 'metrics'
    AND region = 'eu-west-1' AND cell = 'eu-west-1-cell-10' AND silo = 'eu-west-1-
cell-10-silo-1'
    AND availability_zone = 'eu-west-1-3' AND microservice_name = 'demeter'
    AND instance_name = 'i-zaZswmJk-demeter-eu-west-1-cell-10-
silo-1-00000272.amazonaws.com'
GROUP BY BIN(time, 30m)
ORDER BY time_bin desc
```

```
SELECT BIN(time, 30m) AS time_bin,
    AVG(memory_used) AS avg_memory,
    ROUND(APPROX_PERCENTILE(memory_used, 0.99), 2) as p99_memory
FROM "raw_data"."devops"
WHERE time BETWEEN from_milliseconds(1636527099477) AND
 from_milliseconds(1636613499477)
    AND measure_name = 'metrics'
    AND region = 'eu-west-1' AND cell = 'eu-west-1-cell-10' AND silo = 'eu-west-1-
cell-10-silo-1'
    AND availability_zone = 'eu-west-1-3' AND microservice_name = 'demeter'
    AND instance_name = 'i-zaZswmJk-demeter-eu-west-1-cell-10-
silo-1-00000272.amazonaws.com'
GROUP BY BIN(time, 30m)
ORDER BY time_bin desc
```

```
SELECT COUNT(gc_pause)
FROM "raw_data"."devops"
WHERE time BETWEEN from_milliseconds(1636527099477) AND
 from_milliseconds(1636613499478)
    AND measure_name = 'events'
    AND region = 'eu-west-1' AND cell = 'eu-west-1-cell-10' AND silo = 'eu-west-1-
cell-10-silo-1'
    AND availability_zone = 'eu-west-1-3' AND microservice_name = 'demeter'
    AND instance_name = 'i-zaZswmJk-demeter-eu-west-1-cell-10-
silo-1-00000272.amazonaws.com'
```

```
SELECT avg(gc_pause) as avg, round(approx_percentile(gc_pause, 0.99), 2) as p99
FROM "raw_data"."devops"
WHERE time BETWEEN from_milliseconds(1636527099478) AND
 from_milliseconds(1636613499478)
```

```
    AND measure_name = 'events'
    AND region = 'eu-west-1' AND cell = 'eu-west-1-cell-10' AND silo = 'eu-west-1-
cell-10-silo-1'
    AND availability_zone = 'eu-west-1-3' AND microservice_name = 'demeter'
    AND instance_name = 'i-zaZswmJk-demeter-eu-west-1-cell-10-
silo-1-00000272.amazonaws.com'
```

```
SELECT BIN(time, 30m) AS time_bin,
    AVG(disk_io_reads) AS avg,
    ROUND(APPROX_PERCENTILE(disk_io_reads, 0.99), 2) as p99
FROM "raw_data"."devops"
WHERE time BETWEEN from_milliseconds(1636527099478) AND
 from_milliseconds(1636613499478)
    AND measure_name = 'metrics'
    AND region = 'eu-west-1' AND cell = 'eu-west-1-cell-10' AND silo = 'eu-west-1-
cell-10-silo-1'
    AND availability_zone = 'eu-west-1-3' AND microservice_name = 'demeter'
    AND instance_name = 'i-zaZswmJk-demeter-eu-west-1-cell-10-
silo-1-00000272.amazonaws.com'
GROUP BY BIN(time, 30m)
ORDER BY time_bin desc
```

Optimizing costs by sharing scheduled query across dashboards

In this example, we will see a scenario where multiple dashboard panels display variations of similar information (finding high CPU hosts and fraction of fleet with high CPU utilization) and how you can use the same scheduled query to pre-compute results which are then used to populate multiple panels. This reuse further optimizes your costs where instead of using different scheduled queries, one for each panel, you use only owner.

Dashboard panels with raw data

CPU utilization per region per microservice

The first panel computes the instances whose avg CPU utilization is a threshold below or above the above CPU utilization for given deployment within a region, cell, silo, availability zone, and microservice. It then sorts the region and microservice which has the highest percentage of hosts with high utilization. It helps identify how hot the servers of a specific deployment are running, and then subsequently drill down to better understand the issues.

The query for the panel demonstrates the flexibility of Timestream for LiveAnalytics's SQL support to perform complex analytical tasks with common table expressions, window functions, joins, and so on.

Per region, per microservice high CPU utilization hosts

region	microservice_name	num_hosts	high_utilization_hosts	low_utilization_hosts	percent_high_utilization_host	percent_low_utilization_hosts	rank
us-west-2	demeter	2000	430	366	22	18	1
us-east-1	demeter	22500	4625	4455	21	20	1
eu-west-1	demeter	10000	2056	1988	21	20	1
us-east-2	demeter	2000	419	411	21	21	1
ap-northeast-1	demeter	7500	1543	1509	21	20	1
us-west-1	apollo	18000	3651	3637	20	20	1
ap-northeast-1	apollo	22500	4470	4599	20	20	2
eu-west-1	apollo	30000	5994	6036	20	20	2

Query:

```
WITH microservice_cell_avg AS (
    SELECT region, cell, silo, availability_zone, microservice_name, AVG(cpu_user) AS
 microservice_avg_metric
    FROM "raw_data"."devops"
    WHERE time BETWEEN from_milliseconds(1636526593876) AND
 from_milliseconds(1636612993876)
        AND measure_name = 'metrics'
    GROUP BY region, cell, silo, availability_zone, microservice_name
), instance_avg AS (
    SELECT region, cell, silo, availability_zone, microservice_name, instance_name,
        AVG(cpu_user) AS instance_avg_metric
    FROM "raw_data"."devops"
    WHERE time BETWEEN from_milliseconds(1636526593876) AND
 from_milliseconds(1636612993876)
        AND measure_name = 'metrics'
    GROUP BY region, cell, silo, availability_zone, microservice_name, instance_name
), instances_above_threshold AS (
  SELECT i.*,
    CASE WHEN i.instance_avg_metric > (1 + 0.2) * m.microservice_avg_metric THEN 1 ELSE
 0 END AS high_utilization,
    CASE WHEN i.instance_avg_metric < (1 - 0.2) * m.microservice_avg_metric THEN 1 ELSE
 0 END AS low_utilization
  FROM instance_avg i INNER JOIN microservice_cell_avg m
    ON i.region = m.region AND i.cell = m.cell AND i.silo = m.silo AND
 i.availability_zone = m.availability_zone
      AND m.microservice_name = i.microservice_name
), per_deployment_high AS (
```

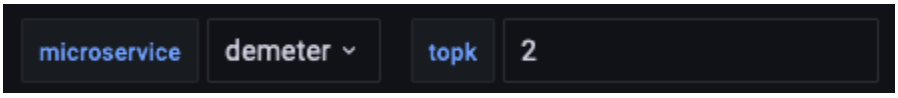
```
SELECT region, microservice_name, COUNT(*) AS num_hosts, SUM(high_utilization) AS
 high_utilization_hosts, SUM(low_utilization) AS low_utilization_hosts,
    ROUND(SUM(high_utilization) * 100.0 / COUNT(*), 0) AS
 percent_high_utilization_hosts,
    ROUND(SUM(low_utilization) * 100.0 / COUNT(*), 0) AS percent_low_utilization_hosts
FROM instances_above_threshold
GROUP BY region, microservice_name
), per_region_ranked AS (
    SELECT *,
        DENSE_RANK() OVER (PARTITION BY region ORDER BY percent_high_utilization_hosts
 DESC, high_utilization_hosts DESC) AS rank
    FROM per_deployment_high
)
SELECT *
FROM per_region_ranked
WHERE rank <= 2
ORDER BY percent_high_utilization_hosts desc, rank asc
```

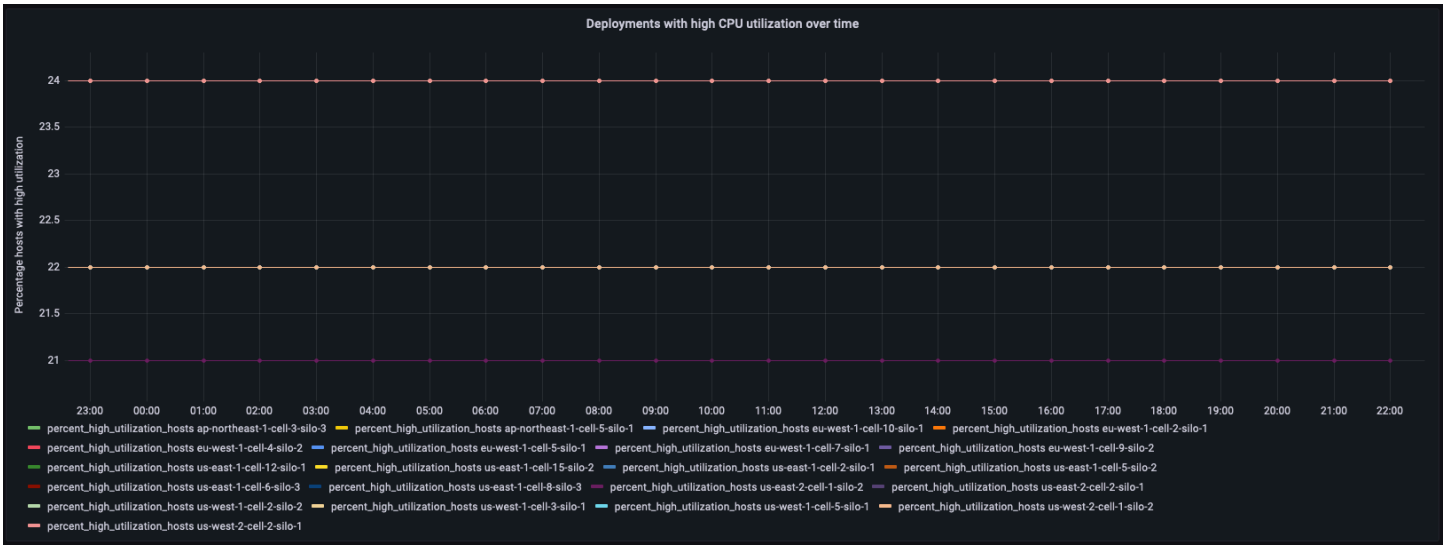
Drill down into a microservice to find hot spots

The next dashboard allows you to drill deeper into one of the microservices to find out the specific region, cell, and silo for that microservice is running what fraction of fraction of its fleet at higher CPU utilization. For instance, in the fleet wide dashboard you saw the microservice demeter show up in the top few ranked positions, so in this dashboard, you want to drill deeper into that microservice.

This dashboard uses a variable to pick microservice to drill down into, and the values of the variable is populated using unique values of the dimension. Once you pick the microservice, the rest of the dashboard refreshes.

As you see below, the first panel plots the percentage of hosts in a deployment (a region, cell, and silo for a microservice) over time, and the corresponding query which is used to plot the dashboard. This plot itself identifies a specific deployment having higher percentage of hosts with high CPU.





Query:

```
WITH microservice_cell_avg AS (
    SELECT region, cell, silo, availability_zone, microservice_name, bin(time, 1h) as
 hour, AVG(cpu_user) AS microservice_avg_metric
    FROM "raw_data"."devops"
    WHERE time BETWEEN from_milliseconds(1636526898831) AND
 from_milliseconds(1636613298831)
        AND measure_name = 'metrics'
        AND microservice_name = 'demeter'
    GROUP BY region, cell, silo, availability_zone, microservice_name, bin(time, 1h)
), instance_avg AS (
    SELECT region, cell, silo, availability_zone, microservice_name, instance_name,
 bin(time, 1h) as hour,
        AVG(cpu_user) AS instance_avg_metric
    FROM "raw_data"."devops"
    WHERE time BETWEEN from_milliseconds(1636526898831) AND
 from_milliseconds(1636613298831)
        AND measure_name = 'metrics'
        AND microservice_name = 'demeter'
    GROUP BY region, cell, silo, availability_zone, microservice_name, instance_name,
 bin(time, 1h)
), instances_above_threshold AS (
  SELECT i.*,
    CASE WHEN i.instance_avg_metric > (1 + 0.2) * m.microservice_avg_metric THEN 1 ELSE
 0 END AS high_utilization
  FROM instance_avg i INNER JOIN microservice_cell_avg m
    ON i.region = m.region AND i.cell = m.cell AND i.silo = m.silo AND
 i.availability_zone = m.availability_zone
```

```
      AND m.microservice_name = i.microservice_name AND m.hour = i.hour
), high_utilization_percent AS (
    SELECT region, cell, silo, microservice_name, hour, COUNT(*) AS num_hosts,
 SUM(high_utilization) AS high_utilization_hosts,
        ROUND(SUM(high_utilization) * 100.0 / COUNT(*), 0) AS
 percent_high_utilization_hosts
    FROM instances_above_threshold
    GROUP BY region, cell, silo, microservice_name, hour
), high_utilization_ranked AS (
    SELECT region, cell, silo, microservice_name,
        DENSE_RANK() OVER (PARTITION BY region ORDER BY
 AVG(percent_high_utilization_hosts) desc, AVG(high_utilization_hosts) desc) AS rank
    FROM high_utilization_percent
    GROUP BY region, cell, silo, microservice_name
)
SELECT hup.silo, CREATE_TIME_SERIES(hour, hup.percent_high_utilization_hosts) AS
 percent_high_utilization_hosts
FROM high_utilization_percent hup INNER JOIN high_utilization_ranked hur
    ON hup.region = hur.region AND hup.cell = hur.cell AND hup.silo = hur.silo AND
 hup.microservice_name = hur.microservice_name
WHERE rank <= 2
GROUP BY hup.region, hup.cell, hup.silo
ORDER BY hup.silo
```

Converting into a single scheduled query enabling reuse

It is important to note that a similar computation is done across the different panels across the two dashboards. You can define a separate scheduled query for each panel. Here you will see how you can further optimize your costs by defining one scheduled query who results can be used to render all the three panels.

Following is the query that captures the aggregates that are computed and used for all the different panels. You will observe several important aspects in the definition of this scheduled query.

- The flexibility and the power of the SQL surface area supported by scheduled queries, where you can use common table expressions, joins, case statements, etc.
- You can using one scheduled query to compute the statistics at a finer granularity than a specific dashboard might need, and for all values that a dashboard might use for different variables. For instance, you will see the aggregates are computed across a region, cell, silo, and microservice. Therefore, you can combine these to create region-level, or region, and microservice-level

aggregates. Similarly, the same query computes the aggregates for all regions, cells, silos, and microservices. It allows you to apply filters on these columns to obtain the aggregates for a subset of the values. For instance, you can compute the aggregates for any one region, say us-east-1, or any one microservice say demeter or drill down into a specific deployment within a region, cell, silo, and microservice. This approach further optimizes your costs of maintaining the pre-computed aggregates.

```
WITH microservice_cell_avg AS (
    SELECT region, cell, silo, availability_zone, microservice_name, bin(time, 1h) as
 hour, AVG(cpu_user) AS microservice_avg_metric
    FROM raw_data.devops
    WHERE time BETWEEN bin(@scheduled_runtime, 1h) - 1h AND bin(@scheduled_runtime, 1h)
 + 1h
        AND measure_name = 'metrics'
    GROUP BY region, cell, silo, availability_zone, microservice_name, bin(time, 1h)

), instance_avg AS (
    SELECT region, cell, silo, availability_zone, microservice_name, instance_name,
 bin(time, 1h) as hour,
        AVG(cpu_user) AS instance_avg_metric
   FROM raw_data.devops
   WHERE time BETWEEN bin(@scheduled_runtime, 1h) - 1h AND bin(@scheduled_runtime, 1h)
 + 1h
       AND measure_name = 'metrics'
   GROUP BY region, cell, silo, availability_zone, microservice_name, instance_name,
 bin(time, 1h)
), instances_above_threshold AS (
    SELECT i.*,
        CASE WHEN i.instance_avg_metric > (1 + 0.2) * m.microservice_avg_metric THEN 1
 ELSE 0 END AS high_utilization,
        CASE WHEN i.instance_avg_metric < (1 - 0.2) * m.microservice_avg_metric THEN 1
 ELSE 0 END AS low_utilization
   FROM instance_avg i INNER JOIN microservice_cell_avg m
       ON i.region = m.region AND i.cell = m.cell AND i.silo = m.silo AND
 i.availability_zone = m.availability_zone
           AND m.microservice_name = i.microservice_name AND m.hour = i.hour
)
SELECT region, cell, silo, microservice_name, hour,
     COUNT(*) AS num_hosts, SUM(high_utilization) AS high_utilization_hosts,
 SUM(low_utilization) AS low_utilization_hosts
FROM instances_above_threshold GROUP BY region, cell, silo, microservice_name, hour
```

The following is a scheduled query definition for the previous query. The schedule expression, it is configured to refresh every 30 mins, and refreshes the data for up to an hour back, again using the bin(@scheduled_runtime, 1h) construct to get the full hour's events. Depending on your application's freshness requirements, you can configure it to refresh more or less frequently. By using WHERE time BETWEEN bin(@scheduled_runtime, 1h) - 1h AND bin(@scheduled_runtime, 1h) + 1h, we can ensure that even if you are refreshing once every 15 minutes, you will get the full hour's data for the current hour and the previous hour.

Later on, you will see how the three panels use these aggregates written to table deployment_cpu_stats_per_hr to visualize the metrics that are relevant to the panel.

```
{
    "Name": "MultiPT30mHighCpuDeploymentsPerHr",
    "QueryString": "WITH microservice_cell_avg AS (    SELECT region, cell,
 silo, availability_zone, microservice_name, bin(time, 1h) as hour, AVG(cpu_user)
 AS microservice_avg_metric    FROM raw_data.devops    WHERE time BETWEEN
 bin(@scheduled_runtime, 1h) - 1h AND bin(@scheduled_runtime, 1h) + 1h    AND
 measure_name = 'metrics'    GROUP BY region, cell, silo, availability_zone,
 microservice_name, bin(time, 1h)    ), instance_avg AS (    SELECT region,
 cell, silo, availability_zone, microservice_name, instance_name, bin(time, 1h)
 as hour,    AVG(cpu_user) AS instance_avg_metric    FROM raw_data.devops
 WHERE time BETWEEN bin(@scheduled_runtime, 1h) - 1h AND bin(@scheduled_runtime,
 1h) + 1h    AND measure_name = 'metrics'    GROUP BY region, cell, silo,
 availability_zone, microservice_name, instance_name, bin(time, 1h)    ),
 instances_above_threshold AS (    SELECT i.*,    CASE WHEN i.instance_avg_metric >
 (1 + 0.2) * m.microservice_avg_metric THEN 1 ELSE 0 END AS high_utilization,    CASE
 WHEN i.instance_avg_metric < (1 - 0.2) * m.microservice_avg_metric THEN 1 ELSE 0 END
 AS low_utilization    FROM instance_avg i INNER JOIN microservice_cell_avg m    ON
 i.region = m.region AND i.cell = m.cell AND i.silo = m.silo AND i.availability_zone
 = m.availability_zone    AND m.microservice_name = i.microservice_name AND m.hour =
 i.hour    )     SELECT region, cell, silo, microservice_name, hour,         COUNT(*)
 AS num_hosts, SUM(high_utilization) AS high_utilization_hosts, SUM(low_utilization) AS
 low_utilization_hosts    FROM instances_above_threshold GROUP BY region, cell, silo,
 microservice_name, hour",
    "ScheduleConfiguration": {
        "ScheduleExpression": "cron(0/30 * * * ? *)"
    },
    "NotificationConfiguration": {
        "SnsConfiguration": {
            "TopicArn": "******"
        }
    },
```

```
    "TargetConfiguration": {
        "TimestreamConfiguration": {
            "DatabaseName": "derived",
            "TableName": "deployment_cpu_stats_per_hr",
            "TimeColumn": "hour",
            "DimensionMappings": [
                {
                    "Name": "region",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "cell",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "silo",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "Name": "microservice_name",
                    "DimensionValueType": "VARCHAR"
                }
            ],
            "MultiMeasureMappings": {
                "TargetMultiMeasureName": "cpu_user",
                "MultiMeasureAttributeMappings": [
                    {
                        "SourceColumn": "num_hosts",
                        "MeasureValueType": "BIGINT"
                    },
                    {
                        "SourceColumn": "high_utilization_hosts",
                        "MeasureValueType": "BIGINT"
                    },
                    {
                        "SourceColumn": "low_utilization_hosts",
                        "MeasureValueType": "BIGINT"
                    }
                ]
            }
        }
    },
    "ErrorReportConfiguration": {
        "S3Configuration" : {
```

```
            "BucketName" : "******",
            "ObjectKeyPrefix": "errors",
            "EncryptionOption": "SSE_S3"
        }
    },
    "ScheduledQueryExecutionRoleArn": "******"
}
```

Dashboard from pre-computed results

High CPU utilization hosts

For the high utilization hosts, you will see how the different panels use the data from deployment_cpu_stats_per_hr to compute different aggregates necessary for the panels. For instance, this panels provides region-level information, so it reports aggregates grouped by region and microservice, without filtering any region or microservice.

Per region, per microservice high utilization hosts

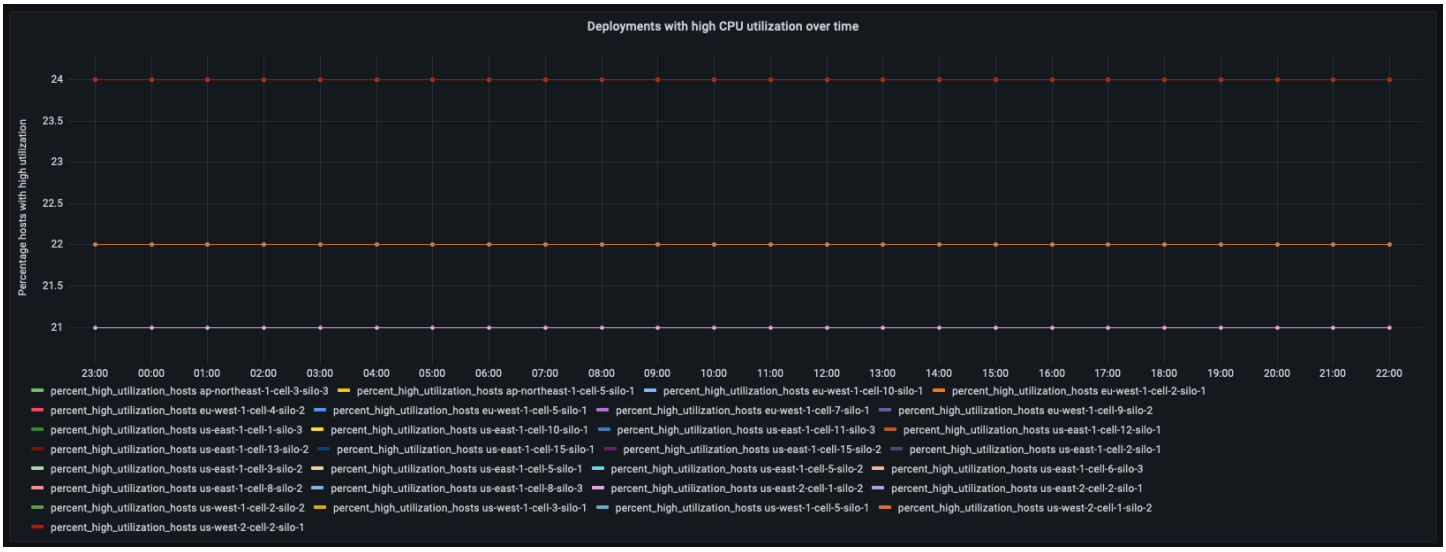
region	microservice_name	num_hosts	high_utilization_hosts	low_utilization_hosts	percent_high_utilization_host	percent_low_utilization_hosts	rank
us-west-2	demeter	1962	423	359	22	18	1
us-east-2	demeter	2000	419	411	21	21	1
us-east-1	demeter	22500	4628	4455	21	20	1
ap-northeast-1	demeter	7500	1544	1509	21	20	1
eu-west-1	demeter	9983	2056	1984	21	20	1
us-west-1	apollo	18000	3657	3643	20	20	1
ap-northeast-1	apollo	22500	4470	4599	20	20	2
us-east-2	hercules	4000	813	752	20	19	2

```
WITH per_deployment_hosts AS (
    SELECT region, cell, silo, microservice_name,
        AVG(num_hosts) AS num_hosts,
        AVG(high_utilization_hosts) AS high_utilization_hosts,
        AVG(low_utilization_hosts) AS low_utilization_hosts
    FROM "derived"."deployment_cpu_stats_per_hr"
    WHERE time BETWEEN from_milliseconds(1636567785437) AND
 from_milliseconds(1636654185437)
        AND measure_name = 'cpu_user'
    GROUP BY region, cell, silo, microservice_name
), per_deployment_high AS (
    SELECT region, microservice_name,
        SUM(num_hosts) AS num_hosts,
        ROUND(SUM(high_utilization_hosts), 0) AS high_utilization_hosts,
        ROUND(SUM(low_utilization_hosts),0) AS low_utilization_hosts,
        ROUND(SUM(high_utilization_hosts) * 100.0 / SUM(num_hosts)) AS
 percent_high_utilization_hosts,
```

```
        ROUND(SUM(low_utilization_hosts) * 100.0 / SUM(num_hosts)) AS
 percent_low_utilization_hosts
    FROM per_deployment_hosts
    GROUP BY region, microservice_name
),
per_region_ranked AS (
    SELECT *,
        DENSE_RANK() OVER (PARTITION BY region ORDER BY percent_high_utilization_hosts
 DESC, high_utilization_hosts DESC) AS rank
    FROM per_deployment_high
)
SELECT *
FROM per_region_ranked
WHERE rank <= 2
ORDER BY percent_high_utilization_hosts desc, rank asc
```

Drill down into a microservice to find high CPU usage deploymentss

This next example again uses the deployment_cpu_stats_per_hr derived table, but now applies a filter for a specific microservice (demeter in this example, since it reported high utilization hosts in the aggregate dashboard). This panel tracks the percentage of high CPU utilization hosts over time.




```
WITH high_utilization_percent AS (
    SELECT region, cell, silo, microservice_name, bin(time, 1h) AS hour, MAX(num_hosts)
 AS num_hosts,
        MAX(high_utilization_hosts) AS high_utilization_hosts,
        ROUND(MAX(high_utilization_hosts) * 100.0 / MAX(num_hosts)) AS
 percent_high_utilization_hosts
    FROM "derived"."deployment_cpu_stats_per_hr"
```

```
    WHERE time BETWEEN from_milliseconds(1636525800000) AND
 from_milliseconds(1636612200000)
        AND measure_name = 'cpu_user'
        AND microservice_name = 'demeter'
    GROUP BY region, cell, silo, microservice_name, bin(time, 1h)
), high_utilization_ranked AS (
    SELECT region, cell, silo, microservice_name,
        DENSE_RANK() OVER (PARTITION BY region ORDER BY
 AVG(percent_high_utilization_hosts) desc, AVG(high_utilization_hosts) desc) AS rank
    FROM high_utilization_percent
    GROUP BY region, cell, silo, microservice_name
)
SELECT hup.silo, CREATE_TIME_SERIES(hour, hup.percent_high_utilization_hosts) AS
 percent_high_utilization_hosts
FROM high_utilization_percent hup INNER JOIN high_utilization_ranked hur
    ON hup.region = hur.region AND hup.cell = hur.cell AND hup.silo = hur.silo AND
 hup.microservice_name = hur.microservice_name
WHERE rank <= 2
GROUP BY hup.region, hup.cell, hup.silo
ORDER BY hup.silo
```

Comparing a query on a base table with a query of scheduled query results

In this Timestream query example, we use the following schema, example queries, and outputs to compare a query on a base table with a query on a derived table of scheduled query results. With a well-planned scheduled query, you can get a derived table with fewer rows and other characteristics that can lead to faster queries than would be possible on the original base table.

For a video that describes this scenario, see Improve query performance and reduce cost using scheduled queries in Amazon Timestream for LiveAnalytics.

For this example, we use the following scenario:

- **Region** – us-east-1
- **Base table** – `"clickstream"."shopping"`
- **Derived table** – `"clickstream"."aggregate"`

Base table

The following describes the schema for the base table.

Column	Type	Timestream for LiveAnalytics attribute type
channel	varchar	MULTI
description	varchar	MULTI
event	varchar	DIMENSION
ip_address	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
product	varchar	MULTI
product_id	varchar	MULTI
quantity	double	MULTI
query	varchar	MULTI
session_id	varchar	DIMENSION
user_group	varchar	DIMENSION
user_id	varchar	DIMENSION

The following describes the measures for the base table. A *base table* refers to a table in Timestream that scheduled query is run on.

- **measure_name** – `metrics`
- **data** – multi
- **dimensions**:

```
[ ( user_group, varchar ),( user_id, varchar ),( session_id, varchar ),( ip_address,
 varchar ),( event, varchar ) ]
```

Query on a base table

The following is an ad-hoc query that gathers counts by a 5-minute aggregate in a given time range.

```
SELECT BIN(time, 5m) as time,
channel,
product_id,
SUM(quantity) as product_quantity
FROM "clickstream"."shopping"
WHERE BIN(time, 5m) BETWEEN '2023-05-11 10:10:00.000000000' AND '2023-05-11
 10:30:00.000000000'
AND channel = 'Social media'
and product_id = '431412'
GROUP BY BIN(time, 5m),channel,product_id
```

Output:

```
duration:1.745 sec
Bytes scanned: 29.89 MB
Query Id: AEBQEANMHG7MHHBHCKJ3BSOE3QUGIDBGWCCP5I6J6YUW5CVJZ2M3JCJ27QRMM7A
Row count:5
```

Scheduled query

The following is a scheduled query that runs every 5 minutes.

```
SELECT BIN(time, 5m) as time, channel as measure_name, product_id, product,
SUM(quantity) as product_quantity
FROM "clickstream"."shopping"
WHERE time BETWEEN BIN(@scheduled_runtime, 5m) - 10m AND BIN(@scheduled_runtime, 5m) -
 5m
AND channel = 'Social media'
GROUP BY BIN(time, 5m), channel, product_id, product
```

Query on a derived table

The following is an ad-hoc query on a derived table. A *derived table* refers to a Timestream table that contains the results of a scheduled query.

```
SELECT time, measure_name, product_id,product_quantity
```

```
FROM "clickstream"."aggregate"
WHERE time BETWEEN '2023-05-11 10:10:00.000000000' AND '2023-05-11 10:30:00.000000000'
AND measure_name = 'Social media'
and product_id = '431412'
```

Output:

```
duration: 0.2960 sec
Bytes scanned: 235.00 B
QueryID: AEBQEANMHHAAQU4FFTT6CFM6UYXTL4SMLZV22MFP4KV2Z7IRVOPLOMLDD6BR33Q
Row count: 5
```

Comparison

The following is a comparison of the results of a query on a base table with a query on a derived table. The same query on a derived table that has aggregated results done through a scheduled query completes faster with fewer scanned bytes.

These results show the value of using scheduled queries to aggregate data for faster queries.

	Query on base table	Query on derived table
Duration	1.745 sec	0.2960 sec
Bytes scanned	29.89 MB	235 bytes
Row count	5	5

Using UNLOAD to export query results to S3 from Timestream for LiveAnalytics

Amazon Timestream for LiveAnalytics now enables you to export your query results to Amazon S3 in a cost-effective and secure way using the UNLOAD statement. Using the UNLOAD statement, you can now export time series data to selected S3 buckets in either Apache Parquet or Comma Separated Values (CSV) format, which provides flexibility to store, combine, and analyze your time series data with other services. The UNLOAD statement allows you to export the data in a compressed manner, which reduces the data transferred and storage space required. UNLOAD

also supports partitioning based on selected attributes when exporting the data, improving performance and reducing the processing time of downstream services accessing the data. In addition, you can use Amazon S3 managed keys (SSE-S3) or AWS Key Management Service (AWS KMS) managed keys (SSE-KMS) to encrypt your exported data.

Benefits of UNLOAD from Timestream for LiveAnalytics

The key benefits of using the `UNLOAD` statement are as follows.

- **Operational ease** – With the `UNLOAD` statement, you can export gigabytes of data in a single query request in either Apache Parquet or CSV format, providing flexibility to select the best suited format for your downstream processing needs and making it easier to build data lakes.
- **Secure and Cost effective** – `UNLOAD` statement provides the capability to export your data to an S3 bucket in a compressed manner and to encrypt (SSE-KMS or SSE_S3) your data using customer managed keys, reducing the data storage costs and protecting against unauthorized access.
- **Performance** – Using the `UNLOAD` statement, you can partition the data when exporting to an S3 bucket. Partitioning the data enables downstream services to process the data in parallel, reducing their processing time. In addition, downstream services can process only the data they need, reducing the processing resources required and thereby costs associated.

Use cases for UNLOAD from Timestream for LiveAnalytics

You can use the `UNLOAD` statement to write data to your S3 bucket to the following.

- **Build Data Warehouse** – You can export gigabytes of query results into S3 bucket and more easily add time series data into your data lake. You can use services such as Amazon Athena and Amazon Redshift to combine your time series data with other relevant data to derive complex business insights.
- **Build AI and ML data pipelines** – The `UNLOAD` statement enables you to easily build data pipelines for your machine learning models that access time series data, making it easier to use time series data with services such as Amazon SageMaker and Amazon EMR.
- **Simplify ETL Processing** – Exporting data into S3 buckets can simplify the process of performing Extract, Transform, Load (ETL) operations on the data, enabling you to seamlessly use third-party tools or AWS services such as AWS Glue to process and transform the data.

UNLOAD Concepts

Syntax

```
UNLOAD (SELECT statement)
 TO 's3://bucket-name/folder'
 WITH ( option = expression [, ...] )
```

where `option` is

```
{ partitioned_by = ARRAY[ col_name[,…] ]
 | format = [ '{ CSV | PARQUET }' ]
 | compression = [ '{ GZIP | NONE }' ]
 | encryption = [ '{ SSE_KMS | SSE_S3 }' ]
 | kms_key = '<string>'
 | field_delimiter ='<character>'
 | escaped_by = '<character>'
 | include_header = ['{true, false}']
 | max_file_size = '<value>'
 | }
```

Parameters

SELECT statement

The query statement used to select and retrieve data from one or more Timestream for LiveAnalytics tables.

```
(SELECT column 1, column 2, column 3 from database.table
      where measure_name = "ABC" and timestamp between ago (1d) and now() )
```

TO clause

```
TO 's3://bucket-name/folder'
```

or

```
TO 's3://access-point-alias/folder'
```

The `TO` clause in the `UNLOAD` statement specifies the destination for the output of the query results. You need to provide the full path, including either Amazon S3 bucket-name or Amazon S3 access-point-alias with folder location on Amazon S3 where Timestream for LiveAnalytics writes the output file objects. The S3 bucket should be owned by the same account and in the same region. In addition to the query result set, Timestream for LiveAnalytics writes the manifest and metadata files to specified destination folder.

PARTITIONED_BY clause

```
partitioned_by = ARRAY [col_name[,…] , (default: none)
```

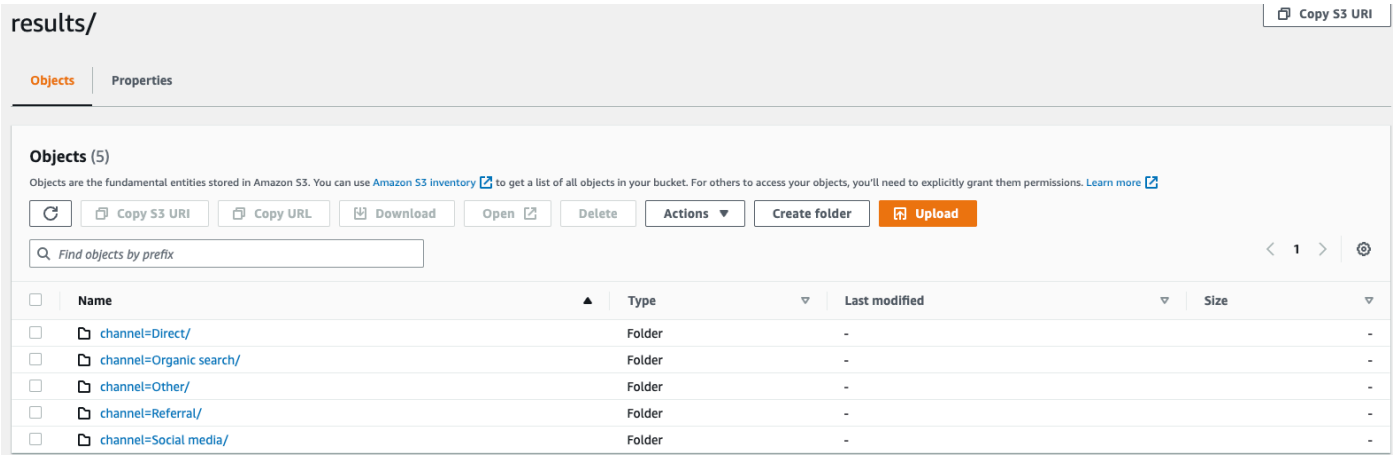
The `partitioned_by` clause is used in queries to group and analyze data at a granular level. When you export your query results to the S3 bucket, you can choose to partition the data based on one or more columns in the select query. When partitioning the data, the exported data is divided into subsets based on the partition column and each subset is stored in a separate folder. Within the results folder that contains your exported data, a sub-folder `folder/results/partition column = partition value/` is automatically created. However, note that partitioned columns are not included in the output file.

`partitioned_by` is not a mandatory clause in the syntax. If you choose to export the data without any partitioning, you can exclude the clause in the syntax.

Example

Assuming you are monitoring clickstream data of your website and have 5 channels of traffic namely `direct`, `Social Media`, `Organic Search`, `Other`, and `Referral`. When exporting the data, you can choose to partition the data using the column `Channel`. Within your data folder, `s3://bucketname/results`, you will have five folders each with their respective channel name, for instance, `s3://bucketname/results/channel=Social Media/`. Within this folder you will find the data of all the customers that landed on your website through the `Social Media` channel. Similarly, you will have other folders for the remaining channels.

Exported data partitioned by Channel column



FORMAT

```
format = [ '{ CSV | PARQUET }' , default: CSV
```

The keywords to specify the format of the query results written to your S3 bucket. You can export the data either as a comma separated value (CSV) using a comma (,) as the default delimiter or in the Apache Parquet format, an efficient open columnar storage format for analytics.

COMPRESSION

```
compression = [ '{ GZIP | NONE }' ], default: GZIP
```

You can compress the exported data using compression algorithm GZIP or have it uncompressed by specifying the `NONE` option.

ENCRYPTION

```
encryption = [ '{ SSE_KMS | SSE_S3 }' ], default: SSE_S3
```

The output files on Amazon S3 are encrypted using your selected encryption option. In addition to your data, the manifest and metadata files are also encrypted based on your selected encryption option. We currently support SSE_S3 and SSE_KMS encryption. SSE_S3 is a server-side encryption with Amazon S3 encrypting the data using 256-bit advanced encryption standard (AES) encryption. SSE_KMS is a server-side encryption to encrypt data using customer-managed keys.

KMS_KEY

```
kms_key = '<string>'
```

KMS Key is a customer-defined key to encrypt exported query results. KMS Key is securely managed by AWS Key Management Service (AWS KMS) and used to encrypt data files on Amazon S3.

FIELD_DELIMITER

```
field_delimiter ='<character>' , default: (,)
```

When exporting the data in CSV format, this field specifies a single ASCII character that is used to separate fields in the output file, such as pipe character (|), a comma (,), or tab (/t). The default delimiter for CSV files is a comma character. If a value in your data contains the chosen delimiter, the delimiter will be quoted with a quote character. For instance, if the value in your data contains `Time,stream`, then this value will be quoted as `"Time,stream"` in the exported data. The quote character used by Timestream for LiveAnalytics is double quotes (").

Avoid specifying the carriage return character (ASCII 13, hex `0D`, text '\r') or the line break character (ASCII 10, hex 0A, text '\n') as the `FIELD_DELIMITER` if you want to include headers in the CSV, since that will prevent many parsers from being able to parse the headers correctly in the resulting CSV output.

ESCAPED_BY

```
escaped_by = '<character>', default: (\)
```

When exporting the data in CSV format, this field specifies the character that should be treated as an escape character in the data file written to S3 bucket. Escaping happens in the following scenarios:

1. If the value itself contains the quote character (") then it will be escaped using an escape character. For example, if the value is `Time"stream`, where (\) is the configured escape character, then it will be escaped as `Time\"stream`.
2. If the value contains the configured escape character, it will be escaped. For example, if the value is `Time\stream`, then it will be escaped as `Time\\stream`.

> **Note**
>
> If the exported output contains complex data type in the like Arrays, Rows or Timeseries, it will be serialized as a JSON string. Following is an example.

Data type	Actual value	How the value is escaped in CSV format [serialized JSON string]
Array	`[ 23,24,25 ]`	`"[23,24,25]"`
Row	`( x=23.0, y=hello )`	`"{\"x\":23.0,\"y\":\"hello\"}"`
Timeseries	`[ ( time=1970-01-01 00:00:00.000000010 , value=100.0 ), ( time=1970-01-01 00:00:00.000000012, value=120.0 ) ]`	`"[{\"time\":\"1970-01-01 00:00:00.000000010Z\",\"value\":100.0},{\"time\":\"1970-01-01 00:00:00.000000012Z\",\"value\":120.0}]"`

INCLUDE_HEADER

```
include_header = 'true' , default: 'false'
```

When exporting the data in CSV format, this field lets you include column names as the first row of the exported CSV data files.

The accepted values are 'true' and 'false' and the default value is 'false'. Text transformation options such as `escaped_by` and `field_delimiter` apply to headers as well.

> **Note**
>
> When including headers, it is important that you not select a carriage return character (ASCII 13, hex 0D, text '\r') or a line break character (ASCII 10, hex 0A, text '\n') as the `FIELD_DELIMITER`, since that will prevent many parsers from being able to parse the headers correctly in the resulting CSV output.

MAX_FILE_SIZE

```
max_file_size = 'X[MB|GB]' , default: '78GB'
```

This field specifies the maximum size of the files that the `UNLOAD` statement creates in Amazon S3. The `UNLOAD` statement can create multiple files but the maximum size of each file written to Amazon S3 will be approximately what is specified in this field.

The value of the field must be between 16 MB and 78 GB, inclusive. You can specify it in integer such as `12GB`, or in decimals such as `0.5GB` or `24.7MB`. The default value is 78 GB.

The actual file size is approximated when the file is being written, so the actual maximum size may not be exactly equal to the number you specify.

What is written to my S3 bucket?

For every successfully executed UNLOAD query, Timestream for LiveAnalytics writes your query results, metadata file and manifest file into the S3 bucket. If you have partitioned the data, you have all the partition folders in the results folder. Manifest file contains a list of the files that were written by the UNLOAD command. Metadata file contains information that describes the characteristics, properties, and attributes of the written data.

What is the exported file name?

The exported file name contains two components, the first component is the queryID and the second component is a unique identifier.

CSV files

```
S3://bucket_name/results/<queryid>_<UUID>.csv
```

```
S3://bucket_name/results/<partitioncolumn>=<partitionvalue>/<queryid>_<UUID>.csv
```

Compressed CSV file

```
S3://bucket_name/results/<partitioncolumn>=<partitionvalue>/<queryid>_<UUID>.gz
```

Parquet file

```
S3://bucket_name/results/<partitioncolumn>=<partitionvalue>/<queryid>_<UUID>.parquet
```

Metadata and Manifest files

```
S3://bucket_name/<queryid>_<UUID>_manifest.json
S3://bucket_name/<queryid>_<UUID>_metadata.json
```

As the data in CSV format is stored at a file level, when you compress the data when exporting to S3, the file will have a “.gz” extension. However, the data in Parquet is compressed at column level so even when you compress the data while exporting, the file will still have .parquet extension.

What information does each file contain?

Manifest file

The manifest file provides information on the list of files that are exported with the UNLOAD execution. The manifest file is available in the provided S3 bucket with a file name: `s3://<bucket_name>/<queryid>_<UUID>_manifest.json`. The manifest file will contain the url of the files in the results folder, the number of records and size of the respective files, and the query metadata (which is total bytes and total rows exported to S3 for the query).

```
{
  "result_files": [
    {
        "url":"s3://my_timestream_unloads/ec2_metrics/
AEDAGANLHLBH4OLISD3CVOZZRWPX5GV2XCXRBKCVD554N6GWPWWXBP7LSG74V2Q_1448466917_szCL4YgVYzGXj2lS.gz"
        "file_metadata":
            {
                "content_length_in_bytes": 32295,
                "row_count": 10
            }
```

```
    },
    {
        "url":"s3://my_timestream_unloads/ec2_metrics/
AEDAGANLHLBH4OLISD3CVOZZRWPX5GV2XCXRBKCVD554N6GWPWWXBP7LSG74V2Q_1448466917_szCL4YgVYzGXj2lS.gz"
        "file_metadata":
            {
                "content_length_in_bytes": 62295,
                "row_count": 20
            }
    },
  ],
  "query_metadata":
    {
      "content_length_in_bytes": 94590,
      "total_row_count": 30,
      "result_format": "CSV",
      "result_version": "Amazon Timestream version 1.0.0"
    },
  "author": {
        "name": "Amazon Timestream",
        "manifest_file_version": "1.0"
  }
}
```

Metadata

The metadata file provides additional information about the data set such as column name, column type, and schema. The metadata file is available in the provided S3 bucket with a file name: S3://bucket_name/<queryid>_<UUID>_metadata.json

Following is an example of a metadata file.

```
{
    "ColumnInfo": [
        {
            "Name": "hostname",
            "Type": {
                "ScalarType": "VARCHAR"
            }
        },
        {
            "Name": "region",
            "Type": {
```

```
                "ScalarType": "VARCHAR"
            }
        },
        {
            "Name": "measure_name",
            "Type": {
                "ScalarType": "VARCHAR"
            }
        },
        {
            "Name": "cpu_utilization",
            "Type": {
                "TimeSeriesMeasureValueColumnInfo": {
                    "Type": {
                        "ScalarType": "DOUBLE"
                    }
                }
            }
        }
  ],
  "Author": {
        "Name": "Amazon Timestream",
        "MetadataFileVersion": "1.0"
  }
}
```

The column information shared in the metadata file has same structure as `ColumnInfo` sent in Query API response for `SELECT` queries.

Results

Results folder contains your exported data in either Apache Parquet or CSV format.

Example

When you submit an `UNLOAD` query like below via Query API,

```
UNLOAD(SELECT user_id, ip_address, event, session_id, measure_name, time, query,
 quantity, product_id, channel
                    FROM sample_clickstream.sample_shopping WHERE time BETWEEN ago(2d)
 AND now())
                TO 's3://my_timestream_unloads/withoutpartition/' WITH ( format='CSV',
 compression='GZIP')
```

UNLOAD query response will have 1 row * 3 columns. Those 3 columns are:

- rows of type BIGINT - indicating the number of rows exported
- metadataFile of type VARCHAR - which is the S3 URI of metadata file exported
- manifestFile of type VARCHAR - which is the S3 URI of manifest file exported

You will get the following response from Query API:

```
{
    "Rows": [
        {
            "Data": [
                {
                    "ScalarValue": "20" # No of rows in output across all files
                },
                {
                    "ScalarValue": "s3://my_timestream_unloads/withoutpartition/
AEDAAANGH3D7FYHOBQGQQMEAISCJ45B42OWWJMOT4N6RRJICZUA7R25VYVOHJIY_<UUID>_metadata.json"
 #Metadata file
                },
                {
                    "ScalarValue": "s3://my_timestream_unloads/withoutpartition/
AEDAAANGH3D7FYHOBQGQQMEAISCJ45B42OWWJMOT4N6RRJICZUA7R25VYVOHJIY_<UUID>_manifest.json"
 #Manifest file
                }
            ]
        }
    ],
    "ColumnInfo": [
        {
            "Name": "rows",
            "Type": {
                "ScalarType": "BIGINT"
            }
        },
        {
            "Name": "metadataFile",
            "Type": {
                "ScalarType": "VARCHAR"
            }
        },
        {
```

```
            "Name": "manifestFile",
            "Type": {
                "ScalarType": "VARCHAR"
            }
        }
    ],
    "QueryId": "AEDAAANGH3D7FYHOBQGQQMEAISCJ45B42OWWJMOT4N6RRJICZUA7R25VYVOHJIY",
    "QueryStatus": {
        "ProgressPercentage": 100.0,
        "CumulativeBytesScanned": 1000,
        "CumulativeBytesMetered": 10000000
    }
}
```

Data types

The UNLOAD statement supports all data types of Timestream for LiveAnalytics's query language described in Supported data types except `time` and unknown.

Prerequisites for UNLOAD from Timestream for LiveAnalytics

Following are prerequisites for writing data to S3 using UNLOAD from Timestream for LiveAnalytics.

- You must have permission to read data from the Timestream for LiveAnalytics table(s) to be used in an UNLOAD command.
- You must have an Amazon S3 bucket in the same AWS Region as your Timestream for LiveAnalytics resources.
- For the selected S3 bucket, ensure that the S3 bucket policy also has permissions to allow Timestream for LiveAnalytics to export the data.
- The credentials used to execute UNLOAD query must have necessary AWS Identity and Access Management (IAM) permissions that allows Timestream for LiveAnalytics to write the data to S3. An example policy would be as follows:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [{
            "Effect": "Allow",
```

```
            "Action": [
                "timestream:Select",
                "timestream:ListMeasures",
                "timestream:WriteRecords",
                "timestream:Unload"
            ],
            "Resource": "arn:aws:timestream:us-
east-2:111122223333:database/database_name/table/table_name"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetBucketAcl",
                "s3:PutObject",
                "s3:GetObject",
                "s3:AbortMultipartUpload"
            ],
            "Resource": [
                "arn:aws:s3:::S3_Bucket_Created",
                "arn:aws:s3:::S3_Bucket_Created/*"
            ]
        }
    ]
}
```

For additional context on these S3 write permissions, refer to the Amazon Simple Storage Service guide. If you are using a KMS key for encrypting the exported data, see the following for the additional IAM policies required.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
            "kms:DescribeKey",
            "kms:Decrypt",
            "kms:GenerateDataKey*"
        ],
```

```
        "Resource": "arn:aws:kms:us-east-2:111122223333:key/*",
        "Condition": {
            "ForAnyValue:StringLike": {
                "kms:ResourceAliases": "alias/Alias_For_Generated_Key"
            }
        }
    }, {
        "Effect": "Allow",
        "Action": [
            "kms:CreateGrant"
        ],
        "Resource": "arn:aws:kms:us-east-2:111122223333:key/*",
        "Condition": {
            "ForAnyValue:StringEquals": {
                "kms:EncryptionContextKeys": "aws:timestream:database_name"
            },
            "Bool": {
                "kms:GrantIsForAWSResource": true
            },
            "StringLike": {
                "kms:ViaService": "timestream.us-east-2.amazonaws.com"
            },
            "ForAnyValue:StringLike": {
                "kms:ResourceAliases": "alias/Alias_For_Generated_Key"
            }
        }
    }
]
}
```

Best practices for UNLOAD from Timestream for LiveAnalytics

Following are best practices related to the UNLOAD command.

- The amount of data that can be exported to S3 bucket using the `UNLOAD` command is not bounded. However, the query times out in 60 minutes and we recommend exporting no more than 60GB of data in a single query. If you need to export more than 60GB of data, split the job across multiple queries.
- While you can send thousands of requests to S3 to upload the data, it is recommended to parallelize the write operations to multiple S3 prefixes. Refer to documentation here. S3 API call rate could be throttled when multiple readers/writers access the same folder.

- Given the limit on S3 key length for defining a prefix, we recommend having bucket and folder names within 10-15 characters, especially when using `partitioned_by` clause.
- When you receive a 4XX or 5XX for queries containing the `UNLOAD` statement, it is possible that partial results are written into the S3 bucket. Timestream for LiveAnalytics does not delete any data from your bucket. Before executing another `UNLOAD` query with same S3 destination, we recommend to manually delete the files created by the failed query. You can identify the files written by a failed query with the corresponding `QueryExecutionId`. For failed queries, Timestream for LiveAnalytics does not export a manifest file to the S3 bucket.
- Timestream for LiveAnalytics uses multi-part upload to export query results to S3. When you receive a 4XX or 5XX from Timestream for LiveAnalytics for queries containing an UNLOAD statement, Timestream for LiveAnalytics does a best-effort abortion of multi-part upload but it is possible that some incomplete parts are left behind. Hence, we recommended to set up an auto cleanup of incomplete multi-part uploads in your S3 bucket by following the guidelines here.

Recommendations for accessing the data in CSV format using CSV parser

- CSV parsers don't allow you to have same character in delimiter, escape, and quote character.
- Some CSV parsers cannot interpret complex data types such as Arrays, we recommend interpreting those through JSON deserializer.

Recommendations for accessing the data in Parquet format

1. If your use case requires UTF-8 character support in schema aka column name, we recommend using Parquet-mr library.
2. The timestamp in your results is represented as a 12 byte integer (INT96)
3. Timeseries will be represented as `array<row<time, value>>`, other nested structures will use corresponding datatypes supported in Parquet format

Using partition_by clause

- The column used in the `partitioned_by` field should be the last column in the select query. If more than one column is used in the `partitioned_by` field, the columns should be the last columns in the select query and in the same order as used in the `partition_by` field.

- The column values used to partition the data (`partitioned_by` field) can contain only ASCII characters. While Timestream for LiveAnalytics allows UTF-8 characters in the values, S3 supports only ASCII characters as object keys.

Example use case for UNLOAD from Timestream for LiveAnalytics

Assume you are monitoring user session metrics, traffic sources, and product purchases of your e-commerce website. You are using Timestream for LiveAnalytics to derive real-time insights into user behavior, product sales, and perform marketing analytics on traffic channels (organic search, social media, direct traffic, paid campaigns and others) that drive customers to the website.

Topics

- Exporting the data without any partitions
- Partitioning data by channel
- Partitioning data by event
- Partitioning data by both channel and event
- Manifest and metadata files
- Using Glue crawlers to build Glue Data Catalog

Exporting the data without any partitions

You want to export the last two days of your data in CSV format.

```
UNLOAD(SELECT user_id, ip_address, event, session_id, measure_name, time,
query, quantity, product_id, channel
FROM sample_clickstream.sample_shopping
WHERE time BETWEEN ago(2d) AND now())
TO 's3://<bucket_name>/withoutpartition'
WITH (  format='CSV',
compression='GZIP')
```

Partitioning data by channel

You want to export the last two days of data in CSV format but would like to have the data from each traffic channel in a separate folder. To do this, you need to partition the data using the `channel` column as shown in the following.

```
UNLOAD(SELECT user_id, ip_address, event, session_id, measure_name, time,
query, quantity, product_id, channel
FROM sample_clickstream.sample_shopping
WHERE time BETWEEN ago(2d) AND now())
TO 's3://<bucket_name>/partitionbychannel/'
WITH (
partitioned_by = ARRAY ['channel'],
format='CSV',
compression='GZIP')
```

Partitioning data by event

You want to export the last two days of data in CSV format but would like to have the data for each event in a separate folder. To do this, you need to partition the data using the `event` column as shown in the following.

```
UNLOAD(SELECT user_id, ip_address, channel, session_id, measure_name, time,
query, quantity, product_id, event
FROM sample_clickstream.sample_shopping
WHERE time BETWEEN ago(2d) AND now())
TO 's3://<bucket_name>/partitionbyevent/'
WITH (
partitioned_by = ARRAY ['event'],
format='CSV',
compression='GZIP')
```

Partitioning data by both channel and event

You want to export the last two days of data in CSV format but would like to have the data for each channel and within channel store each event in a separate folder. To do this, you need to partition the data using both `channel` and `event` column as shown in the following.

```
UNLOAD(SELECT user_id, ip_address, session_id, measure_name, time,
query, quantity, product_id, channel,event
FROM sample_clickstream.sample_shopping
WHERE time BETWEEN ago(2d) AND now())
TO 's3://<bucket_name>/partitionbychannelevent/'
WITH (
partitioned_by = ARRAY ['channel','event'],
format='CSV',
compression='GZIP')
```

Manifest and metadata files

Manifest file

The manifest file provides information on the list of files that are exported with the UNLOAD execution. The manifest file is available in the provided S3 bucket with a file name: `S3://bucket_name/<queryid>_<UUID>_manifest.json`. The manifest file will contain the url of the files in the results folder, the number of records and size of the respective files, and the query metadata (which is total bytes and total rows exported to S3 for the query).

```
{
  "result_files": [
    {
        "url":"s3://my_timestream_unloads/ec2_metrics/
AEDAGANLHLBH4OLISD3CVOZZRWPX5GV2XCXRBKCVD554N6GWPWWXBP7LSG74V2Q_1448466917_szCL4YgVYzGXj2lS.gz"
        "file_metadata":
            {
                "content_length_in_bytes": 32295,
                "row_count": 10
            }
    },
    {
        "url":"s3://my_timestream_unloads/ec2_metrics/
AEDAGANLHLBH4OLISD3CVOZZRWPX5GV2XCXRBKCVD554N6GWPWWXBP7LSG74V2Q_1448466917_szCL4YgVYzGXj2lS.gz"
        "file_metadata":
            {
                "content_length_in_bytes": 62295,
                "row_count": 20
            }
    },
  ],
  "query_metadata":
    {
      "content_length_in_bytes": 94590,
      "total_row_count": 30,
      "result_format": "CSV",
      "result_version": "Amazon Timestream version 1.0.0"
    },
  "author": {
        "name": "Amazon Timestream",
        "manifest_file_version": "1.0"
  }
```

```
}
```

Metadata

The metadata file provides additional information about the data set such as column name, column type, and schema. The metadata file is available in the provided S3 bucket with a file name: S3://bucket_name/<queryid>_<UUID>_metadata.json

Following is an example of a metadata file.

```
{
    "ColumnInfo": [
        {
            "Name": "hostname",
            "Type": {
                "ScalarType": "VARCHAR"
            }
        },
        {
            "Name": "region",
            "Type": {
                "ScalarType": "VARCHAR"
            }
        },
        {
            "Name": "measure_name",
            "Type": {
                "ScalarType": "VARCHAR"
            }
        },
        {
            "Name": "cpu_utilization",
            "Type": {
                "TimeSeriesMeasureValueColumnInfo": {
                    "Type": {
                        "ScalarType": "DOUBLE"
                    }
                }
            }
        }
  ],
  "Author": {
        "Name": "Amazon Timestream",
```

```
        "MetadataFileVersion": "1.0"
  }
}
```

The column information shared in the metadata file has same structure as `ColumnInfo` sent in Query API response for `SELECT` queries.

Using Glue crawlers to build Glue Data Catalog

1. Login to your account with Admin credentials for the following validation.
2. Create a Crawler for Glue Database using the guidelines provided here. Please note that the S3 folder to be provided in the datasource should be the `UNLOAD` result folder such as `s3://my_timestream_unloads/results`.
3. Run the crawler following the guidelines here.
4. View the Glue table.
   - Go to **AWS Glue** → **Tables**.
   - You will see a new table created with table prefix provided while creating the crawler.
   - You can see the schema and partition information by clicking the table details view.

The following are other AWS services and open-source projects that use the AWS Glue Data Catalog.

- **Amazon Athena** – For more information, see Understanding tables, databases, and data catalogs in the Amazon Athena User Guide.
- **Amazon Redshift Spectrum** – For more information, see Querying external data using Amazon Redshift Spectrum in the Amazon Redshift Database Developer Guide.
- **Amazon EMR** – For more information, see Use resource-based policies for Amazon EMR access to AWS Glue Data Catalog in the Amazon EMR Management Guide.
- **AWS Glue Data Catalog client for Apache Hive metastore** – For more information about this GitHub project, see AWS Glue Data Catalog Client for Apache Hive Metastore.

Limits for UNLOAD from Timestream for LiveAnalytics

Following are limits related to the `UNLOAD` command.

- Concurrency for queries using the `UNLOAD` statement is 1 query per second (QPS). Exceeding the query rate might result in throttling.
- Queries containing `UNLOAD` statement can export at most 100 partitions per query. We recommend to check the distinct count of the selected column before using it to partition the exported data.
- Queries containing `UNLOAD` statement time out after 60 minutes.
- The maximum size of the files that the `UNLOAD` statement creates in Amazon S3 is 78 GB.

For other limits for Timestream for LiveAnalytics, see Quotas

Using query insights to optimize queries in Amazon Timestream

Query insights is a performance tuning feature that helps you optimize your queries, improve their performance, and reduce costs. With query insights, you can assess the temporal, time-based, and spatial partition key-based pruning efficiency of your queries. Using query insights, you can also identify areas for improvement to enhance query performance. In addition, with query insights, you can evaluate how effectively your queries use time-based and partition key-based indexing to optimize data retrieval. To optimize query performance, it's essential to fine-tune both the temporal and spatial parameters that govern query execution.

Topics

- Benefits of query insights
- Optimizing data access in Amazon Timestream
- Enabling query insights in Amazon Timestream
- Optimizing queries using query insights response

Benefits of query insights

The following are the key benefits of using query insights:

- **Identifying inefficient queries** – Query insights provides information on the time-based and attribute-based pruning of the tables accessed by the query. This information helps you identify the tables that are sub-optimally accessed.

- **Optimizing your data model and partitioning** – You can use the query insights information to access and fine-tune your data model and partitioning strategy.
- **Tuning queries** – Query insights highlights opportunities to use indexes more effectively.

Optimizing data access in Amazon Timestream

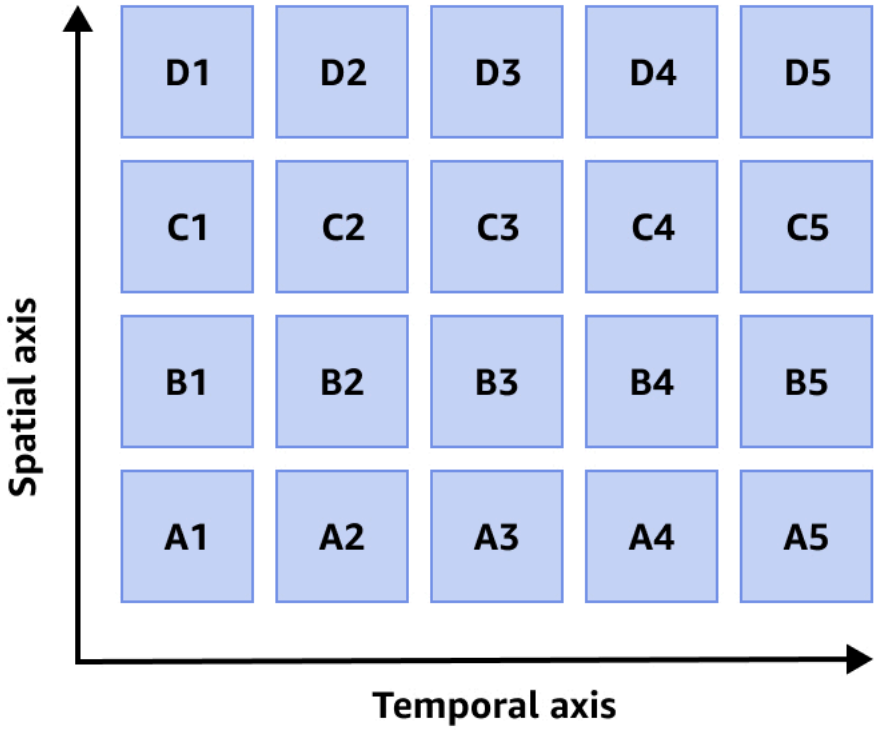
You can optimize the data access patterns in Amazon Timestream using the Timestream partitioning scheme or data organization techniques.

Topics

- Timestream partitioning scheme
- Data organization

Timestream partitioning scheme

Amazon Timestream uses a highly scalable partitioning scheme where each Timestream table can have hundreds, thousands, or even millions of independent partitions. A highly available partition tracking and indexing service manages the partitioning, minimizing the impact of failures and making the system more resilient.



Data organization

Timestream stores each data point it ingests in a single partition. As you ingest data into a Timestream table, Timestream automatically creates partitions based on the timestamps, partition key, and other context attributes in the data. In addition to partitioning the data on time (temporal partitioning), Timestream also partitions the data based on the selected partitioning key and other dimensions (spatial partitioning). This approach is designed to distribute write traffic and allow for effective pruning of data for queries.

The query insights feature provides valuable insights into the pruning efficiency of the query, which includes query spatial coverage and query temporal coverage.

Topics

- QuerySpatialCoverage
- QueryTemporalCoverage

QuerySpatialCoverage

The QuerySpatialCoverage metric provides insights into the spatial coverage of the executed query and the table with the most inefficient spatial pruning. This information can help you identify areas of improvement in the partitioning strategy to enhance spatial pruning. The value for the `QuerySpatialCoverage` metric ranges between 0 and 1. The lower the value of the metric, the more optimal the query pruning on the spatial axis. For example, a value of 0.1 indicates that the query scans 10% of the spatial axis. A value of 1 indicates that the query scans 100% of the spatial axis.

Example Using query insights to analyze a query's spatial coverage

Say that you've a Timestream database that stores weather data. Assume that the temperature is recorded every hour from weather stations located across different states in United States. Imagine that you choose `State` as the customer-defined partitioning key (CDPK) to partition the data by state.

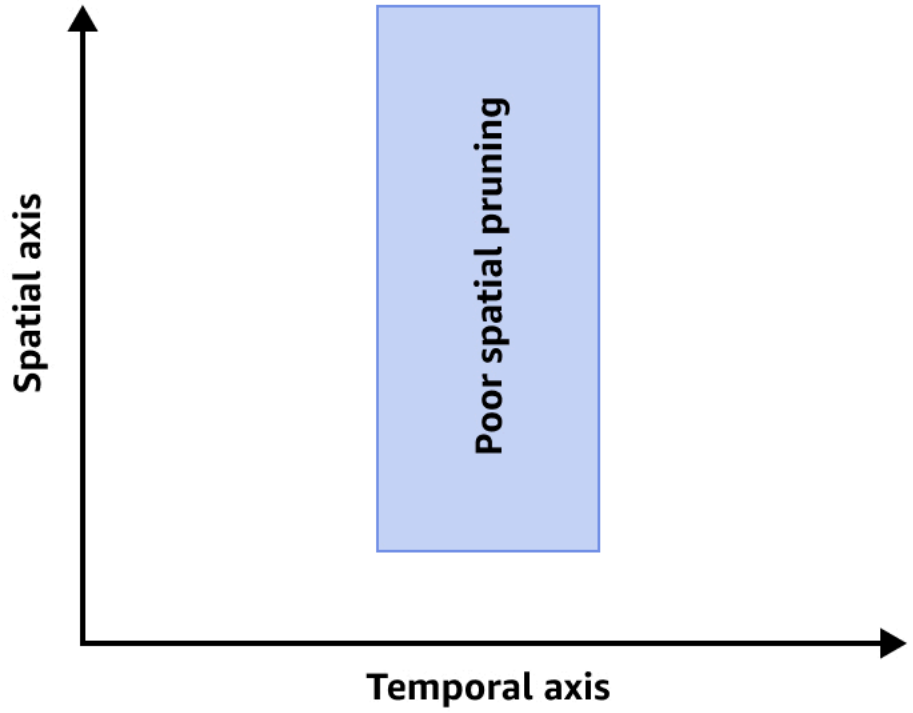
Suppose that you execute a query to retrieve the average temperature for all weather stations in California between 2 PM and 4 PM on a specific day. The following example shows the query for this scenario.

```
SELECT AVG(temperature)
FROM "weather_data"."hourly_weather"
WHERE time >= '2024-10-01 14:00:00' AND time < '2024-10-01 16:00:00'
  AND state = 'CA';
```

Using the query insights feature, you can analyze the query's spatial coverage. Imagine that the `QuerySpatialCoverage` metric returns a value of 0.02. This means that the query only scanned 2% of the spatial axis, which is efficient. In this case, the query was able to effectively prune the spatial range, only retrieving data from California and ignoring data from other states.

On the contrary, if the `QuerySpatialCoverage` metric returned a value of 0.8, it would indicate that the query scanned 80% of the spatial axis, which is less efficient. This might suggest that the partitioning strategy needs to be refined to improve spatial pruning. For example, you can select the partition key as city or region instead of a state. By analyzing the `QuerySpatialCoverage` metric, you can identify opportunities to optimize your partitioning strategy and improve the performance of your queries.

The following image shows poor spatial pruning.



To improve spatial pruning efficiency, you can do one or both of the following:

- Add `measure_name`, the default paritioning key, or use the CDPK predicates in your query.
- If you've already added the attributes mentioned in the previous point, remove functions around these attributes or clauses, such as `LIKE`.

QueryTemporalCoverage

The `QueryTemporalCoverage` metric provides insights into the temporal range scanned by the executed query, including the table with the largest time range scanned. The value for the `QueryTemporalCoverage` metric is time range represented in nanoseconds. The lower the value of this metric, the more optimal the query pruning on the temporal range. For example, a query scanning last few minutes of data is more performant than a query scanning the entire time range of the table.

Example

Say that you've a Timestream database that stores IoT sensor data, with measurements taken every minute from devices located in a manufacturing plant. Assume that you've partitioned your data by `device_ID`.

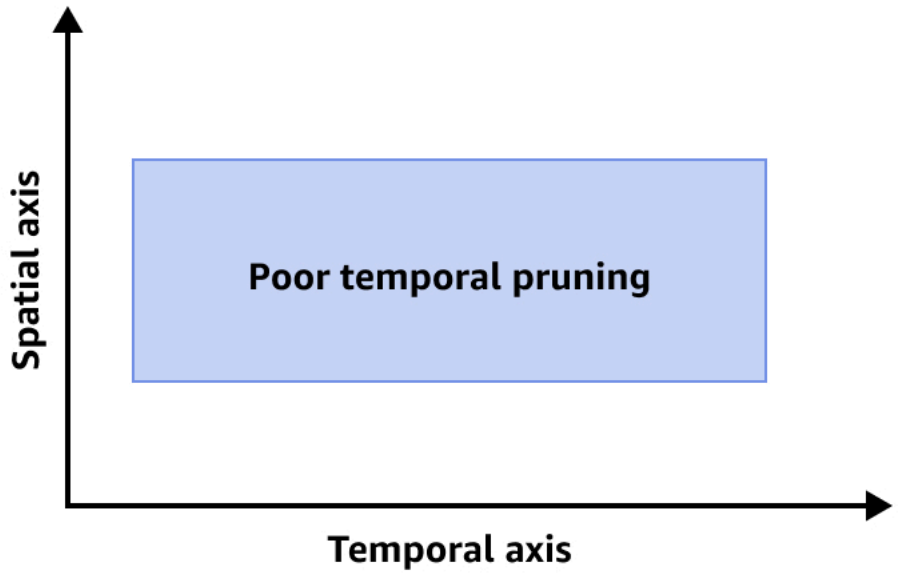
Suppose that you execute a query to retrieve the average sensor reading for a specific device over the last 30 minutes. The following example shows the query for this scenario.

```
SELECT AVG(sensor_reading)
FROM "sensor_data"."factory_1"
WHERE device_id = 'DEV_123'
  AND time >= NOW() - INTERVAL 30 MINUTE and time < NOW();
```

Using the query insights feature, you can analyze the temporal range scanned by the query. Imagine the `QueryTemporalCoverage` metric returns a value of 1800000000000 nanoseconds (30 minutes). This means that the query only scanned the last 30 minutes of data, which is a relatively narrow temporal range. This is a good sign because it indicates that the query was able to effectively prune the temporal partitioning and only retrieved the requested data.

On the contrary, if the `QueryTemporalCoverage` metric returned a value of 1 year in nanoseconds, it indicates that the query scanned one year of time range in the table, which is less efficient. This might suggest that the query is not optimized for temporal pruning, and you could improve it by adding time filters.

The following image shows poor temporal pruning.



To improve temporal pruning, we recommend that you do one or all of the following:

- Add the missing time predicates in the query and make sure that the time predicates are pruning the desired time window.
- Remove functions, such as `MAX()`, around the time predicates.
- Add time predicates to all the sub queries. This is important if your sub queries are joining large tables or performing complex operations.

Enabling query insights in Amazon Timestream

You can enable query insights for your queries with insights delivered directly through the query response. Enabling query insights doesn't require additional infrastructure or incur any additional costs. When you enable query insights, it returns query performance related metadata fields in addition to query results as part of your query response. You can use this information to tune your queries to improve query performance and reduce query cost.

For information about enabling query insights, see Run a query.

To view examples of the responses returned by enabling query insights, see Examples for scheduled queries.

> **Note**
>
> - When you enable query insights, it rate limits the query to 1 query per second (QPS). To avoid performance impacts, we strongly recommend that you enable query insights only during the evaluation phase of your queries, before deploying them to production.
> - The insights provided in query insights are eventually consistent, which means they might change as new data is continuously ingested into the tables.

Optimizing queries using query insights response

Say that you're using Amazon Timestream for LiveAnalytics to monitor energy consumption across various locations. Imagine that you've two tables in your database named `raw-metrics` and `aggregate-metrics`.

The `raw-metrics` table stores detailed energy data at the device level and contains the following columns:

- Timestamp
- State, for example, Washington
- Device ID
- Energy consumption

The data for this table is collected and stored at a minute-by-minute granularity. The table uses `State` as the CDPK.

The `aggregate-metrics` table stores the result of a scheduled query to aggregate the energy consumption data across all devices hourly. This table contains the following columns:

- Timestamp
- State, for example, Washington
- Total energy consumption

The `aggregate-metrics` table stores this data at an hourly granularity. The table uses `State` as the CDPK.

Topics

- Querying energy consumption for the last 24 hours
- Optimizing the query for temporal range
- Optimizing the query for spatial coverage
- Improved query performance

Querying energy consumption for the last 24 hours

Say that you want to extract the total energy consumed in Washington over the last 24 hours. To find this data, you can leverage the strengths of both the tables: `raw-metrics` and `aggregate-metrics`. The `aggregate-metrics` table provides hourly energy consumption data for the last 23 hours, while the `raw-metrics` table offers minute-granular data for the last one hour. By querying across both tables, you can get a complete and accurate picture of energy consumption in Washington over the last 24 hours.

```
SELECT am.time, am.state, am.total_energy_consumption,
rm.time, rm.state, rm.device_id, rm.energy_consumption
FROM
 "metrics"."aggregate-metrics" am
 LEFT JOIN "metrics"."raw-metrics" rm ON am.state = rm.state
WHERE rm.time >= ago(1h) and rm.time < now()
```

This example query is provided for illustrative purposes only and might not work as is. It's intended to demonstrate the concept, but you might need to modify it to fit your specific use case or environment.

After executing this query, you might notice that the query response time is slower than expected. To identify the root cause of this performance issue, you can use the query insights feature to analyze the query's performance and optimize its execution.

The following example shows the query insights response.

```
queryInsightsResponse={
                QuerySpatialCoverage: {
                    Max: {
                        Value: 1.0,
                        TableArn: arn:aws:timestream:us-east-1:123456789012:database/
metrics/table/raw-metrics,
                        PartitionKey: [State]
```

```
                    }
                },
                QueryTemporalRange: {
                    Max: {
                        Value:31540000000000000 //365 days,
                        TableArn: arn:aws:timestream:us-east-1:123456789012:database/
metrics/table/aggregate-metrics
                    }
                },
                QueryTableCount: 2,
                OutputRows: 83,
                OutputBytes: 590
```

The query insights response provides the following information:

- **Temporal range**: The query scanned an excessive 365-day temporal range for the `aggregate-metrics` table. This indicates an inefficient use of temporal filtering.
- **Spatial coverage**: The query scanned the entire spatial range (100%) of the `raw-metrics` table. This suggests that the spatial filtering isn't being utilized effectively.

If your query accesses more than one table, query insights provides the metrics for the table with most sub-optimal access pattern.

Optimizing the query for temporal range

Based on the query insights response, you can optimize the query for temporal range as shown in the following example.

```
SELECT am.time, am.state, am.total_energy_consumption,
rm.time, rm.state, rm.device_id, rm.energy_consumption
FROM
  "metrics"."aggregate-metrics" am
  LEFT JOIN "metrics"."raw-metrics" rm ON am.state = rm.state
WHERE
  am.time >=  ago(23h) and am.time < now()
  AND rm.time >=  ago(1h) and rm.time < now()
  AND rm.state = 'Washington'
```

If you run the `QueryInsights` command again, it returns the following response.

```
queryInsightsResponse={
```

```
                QuerySpatialCoverage: {
                    Max: {
                        Value: 1.0,
                        TableArn: arn:aws:timestream:us-east-1:123456789012:database/
metrics/table/aggregate-metrics,
                        PartitionKey: [State]
                    }
                },
                QueryTemporalRange: {
                    Max: {
                        Value: 82800000000000 //23 hours,
                        TableArn: arn:aws:timestream:us-east-1:123456789012:database/
metrics/table/aggregate-metrics
                    }
                },
                QueryTableCount: 2,
                OutputRows: 83,
                OutputBytes: 590
```

This response shows that the spatial coverage for the `aggregate-metrics` table is still 100%, which is inefficient. The following section shows how to optimze the query for spatial coverage.

Optimizing the query for spatial coverage

Based on the query insights response, you can optimize the query for spatial coverage as shown in the following example.

```
SELECT am.time, am.state, am.total_energy_consumption,
rm.time, rm.state, rm.device_id, rm.energy_consumption
FROM
  "metrics"."aggregate-metrics" am
  LEFT JOIN "metrics"."raw-metrics" rm ON am.state = rm.state
WHERE
  am.time >=  ago(23h) and am.time < now()
  AND am.state ='Washington'
  AND rm.time >=  ago(1h) and rm.time < now()
  AND rm.state = 'Washington'
```

If you run the `QueryInsights` command again, it returns the following response.

```
queryInsightsResponse={
                QuerySpatialCoverage: {
```

```
                    Max: {
                        Value: 0.02,
                        TableArn: arn:aws:timestream:us-east-1:123456789012:database/
metrics/table/aggregate-metrics,
                        PartitionKey: [State]
                    }
                },
                QueryTemporalRange: {
                    Max: {
                        Value: 82800000000000 //23 hours,
                        TableArn: arn:aws:timestream:us-east-1:123456789012:database/
metrics/table/aggregate-metrics
                    }
                },
                QueryTableCount: 2,
                OutputRows: 83,
                OutputBytes: 590
```

Improved query performance

After optimizing the query, query insights provides the following information:

- Temporal pruning for the `aggregate-metrics` table is 23 hours. This indicates that only 23 hours of the temporal range is scanned.
- Spatial pruning for `aggregate-metrics` table is 0.02. This indicates that only 2% of the table's spatial range data is being scanned. The query scans a very small portion of the tables leading to fast performance and reduced resource utilization. The improved pruning efficiency indicates that the query is now optimized for performance.

Working with AWS Backup

The data protection functionality in Amazon Timestream for LiveAnalytics is a fully managed solution to help you meet your regulatory compliance and business continuity requirements. The functionality is enabled through native integration with AWS Backup, a unified backup service designed to simplify the creation, migration, restoration, and deletion of backups, while providing improved reporting and auditing. Through integration with AWS Backup, you can use a fully managed, policy-driven centralized data protection solution to create immutable backups and centrally manage data protection of your application data spanning Timestream and other AWS services supported by AWS Backup.

To use the functionality, you must opt-in to allow AWS Backup to protect your Timestream resources. Opt-in choices apply to the specific account and AWS Region, so you might have to opt in to multiple Regions using the same account. For more information on AWS Backup, see the AWS Backup Developer Guide.

Data Protection functionality available through AWS Backup includes the following.

Scheduled backups—You can set up regularly scheduled backups of your Timestream for LiveAnalytics tables using backup plans.

Cross-account and cross-Region copying—You can automatically copy your backups to another backup vault in a different AWS Region or account, which allows you to support your data protection requirements.

Cold storage tiering—You can configure your backups to implement life cycle rules to delete or transition backups to colder storage. This can help you optimize your backup costs.

Tags—You can automatically tag your backups for billing and cost allocation purposes.

Encryption—Your backup data is stored in the AWS Backup vault. This allows you to encrypt and secure your backups by using an AWS KMS key that is independent from your Timestream for LiveAnalytics table encryption key.

Secure backups using the WORM model—You can use AWS Backup Vault Lock to enable a write-once-read-many (WORM) setting for your backups. With AWS Backup Vault Lock, you can add an additional layer of defense that protects backups from inadvertent or malicious delete operations, changes to backup retention periods, and updates to lifecycle settings. To learn more, see AWS Backup Vault Lock.

The data protection functionality is available in all regions To learn more about the functionality, see the AWS Backup Developer Guide.

Backing up and restoring Timestream tables: How it works

You can create backups of your Amazon Timestream tables. This section provides an overview of what happens during the backup and restore process.

Topics

- Backups
- Restores

Backups

You can use the on-demand backup feature to create full backups of your Amazon Timestream for LiveAnalytics tables. This section provides an overview of what happens during the backup and restore process.

You can create a backup of your Timestream data at a table granularity. You can initiate a backup of the selected table using either Timestream console, or AWS Backup console, SDK, or CLI. The backup is created asynchronously and all the data in the table until the backup initiation time is included in the backup. However, there is a possibility that some of the data ingested into the table while the backup is in progress might also be included in the backup. To protect your data, you can either create a one-time on-demand backup or schedule a recurring backup of your table.

While a backup is in progress, you cannot do the following.

- Pause or cancel the backup operation.
- Delete the source table of the backup.
- Disable backups on a table if a backup for that table is in progress.

Once configured, AWS Backup provides automated backup schedules, retention management, and lifecycle management, removing the need for custom scripts and manual processes. For more information, see the AWS Backup Developer Guide

All Timestream for LiveAnalytics backups are incremental in nature, implying that the first backup of a table is a full backup and every subsequent backup of the same table is an incremental backup, copying only the changes to the data since the last backup. As the data in Timestream for LiveAnalytics is stored in a collection of partitions, all the partitions that changed either due to ingesting new data or updates to the existing data since the last backup are copied during subsequent backups.

If you are using Timestream for LiveAnalytics console, the backups created for all the resources in the account are listed in the **Backups** tab. Additionally, the backups are also listed in the **Table** details.

Restores

You can restore a table from the Timestream for LiveAnalytics console, or AWS Backup console, SDK, or AWS CLI. You can either restore the entire data from your backup, or configure the table

retention settings to restore select data. When you initiate a restore, you can configure the following table settings.

- Database Name
- Table Name
- Memory store retention
- Magnetic store retention
- Enable Magnetic storage writes
- S3 error logs location (optional)
- IAM role that AWS Backup will assume when restoring the backup

The preceding configurations are independent of the source table. To restore all the data in your backup, we recommend that you configure the new table settings such that the sum of memory store retention period and magnetic store retention period is greater than the difference between the oldest timestamp and now. When you select a backup that is incremental to restore, all data (incremental + underlying full data) is restored. Upon successful restore, the table is in active state and you can perform ingestion and/or query operations on the restored table. However, you cannot perform these operations while the restore is in progress. Once restored, the table is similar to any other table in your account.

Example Restore the all data from a backup

This example has the following assumptions.

Oldest timestamp—`August 1, 2021 0:00:00`

- *Now*—`November 9, 2022 0:00:00`

To restore all data from a backup, enter and compare values as follows.

1. Enter **Memory store retention** and **Magnetic store retention**. For example, assume these values.
   - *Memory store retention*—12 hours
   - *Magnetic store retention*—500 days
2. Find the sum of **Memory store retention** and **Magnetic store retention**.

```
12 hours + (500 * 24 hours) =
```

```
12 hours + 12,000 hours =
12,012 hours
```

3. Find the difference between **Oldest timestamp** and now.

```
November 9, 2022 0:00:00 - August 1, 2021 0:00:00 =
465 days =
465 * 24 hours =
11,160 hours
```

4. Ensure the sum of retention values in the second step is greater than difference of times in the third step. Adjust the retention times if necessary.

```
12,012 > 11,160
true
```

Example Restore select data from a backup

This example has the following assumption.

- *Now*—`November 9, 2022 0:00:00`

To restore only select data from a backup, enter and compare values as follows.

1. Determine the earliest timestamp required. For example, assume `December 4, 2021 0:00:00`.
2. Find the difference between the earliest timestamp required and now.

```
November 9, 2022 0:00:00 - December 4, 2021 0:00:00 =
340 days =
340 * 24 hours =
8,160 hours
```

3. Enter the desired value for **Memory store retention**. For example, enter 12 hours.
4. Subtract the value from the difference in the second step.

```
8,160 hours - 12 hours =
8148 hours
```

5. Enter that value for **Magnetic store retention**.

You can copy a backup of your Timestream for LiveAnalytics table data to a different AWS Region and then restore it in that new Region. You can copy and then restore backups between AWS commercial Regions, and AWS GovCloud (US) Regions. You pay only for the data you copy from the source Region and the data you restore to a new table in the destination Region.

Once the table is restored, you must manually set up the following on the restored table.

- AWS Identity and Access Management (IAM) policies
- Tags
- Scheduled Queries

Restore times are directly related to the configuration of your tables. These include the size of your tables, the number of underlying partitions, the amount of data restored to memory store, and other variables. A best practice when planning for disaster recovery is to regularly document average restore completion times and establish how these times affect your overall Recovery Time Objective (RTO).

All backup and restore console and API actions are captured and recorded in AWS CloudTrail for logging, continuous monitoring, and auditing.

Creating backups of Amazon Timestream tables

This section describes how to enable AWS Backup and create on-demand and scheduled backups for Amazon Timestream.

Topics

- Enabling AWS Backup to protect Timestream for LiveAnalytics data
- Creating on-demand backups
- Scheduled backups

Enabling AWS Backup to protect Timestream for LiveAnalytics data

You must enable AWS Backup to use it with Timestream for LiveAnalytics.

To enable AWS Backup in the Timestream for LiveAnalytics console, perform the following steps.

1. Sign in to the AWS Management Console.

2. A pop-up banner appears at the top of your Timestream for LiveAnalytics dashboard page to enable AWS Backup to support Timestream for LiveAnalytics data. Otherwise, from the navigation pane, choose **Backups**.
3. In the **Backup** window, you will see the banner to enable AWS Backup. Choose **Enable**.

   Data Protection through AWS Backup is now available for your Timestream for LiveAnalytics tables.

To enable through AWS Backup, refer to AWS Backup documentation to enable via console and programmatically.

If you choose to disable AWS Backup from protection your Timestream for LiveAnalytics data after those have been enabled, log in through AWS Backup console and move the toggle to the left.

If you can't enable or disable the AWS Backup features, your AWS admin may need to perform those actions.

Creating on-demand backups

To create an on-demand backup of a Timestream for LiveAnalytics table, follow these steps.

1. Sign in to the AWS Management Console.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. Choose **Create on-demand backup**.
4. Continue to select the settings in the backup window.
5. You can either create a backup now, initiates a backup immediately, or select a backup window to start the backup.
6. Select the lifecycle management policy of your backup. You can transition your backup data into cold storage where you have to retain the backup for a minimum of 90 days. You can set the required retention period for your backup You can either select an existing vault or or select **create new backup vault** to navigate to AWS Backup console and create a new backup vault <documentation link on creating a new backup vault here>
7. Select the appropriate IAM role.
8. If you want to assign one or more tags to your on-demand backup, enter a **key** and optional **value**, and choose **Add tag**.
9. Choose to create an on-demand backup. This takes you to the **Backup** page, where you will see a list of jobs.

10. Choose the **Backup job ID** for the resource that you chose to back up to see the details of that job.

Scheduled backups

To schedule a backup, refer to Create a scheduled backup.

Restoring a backup of an Amazon Timestream table

This section describes how to restore a backup of an Amazon Timestream table.

Topics

- Restoring a Timestream for LiveAnalytics table from AWS Backup
- Restoring a Timestream for LiveAnalytics table to another Region or account

Restoring a Timestream for LiveAnalytics table from AWS Backup

To restore your Timestream for LiveAnalytics table from AWS Backup using Timestream for LiveAnalytics console, follow these steps.

1. Sign in to the AWS Management Console.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. To restore a resource, choose the radio button next to the recovery point ID of the resource. In the upper-right corner of the pane, choose **Restore**.
4. Enter the table configuration settings, namely **Database name** and **Table Name**. Please note, the restored table name should be different from the original source table name.
5. Configure the memory and magnetic store retention settings.
6. For **Restore role**, choose the IAM role that AWS Backup will assume for this restore.
7. Choose **Restore backup**. A message at the top of the page provides information about the restore job.

> **Note**
>
> You are charged for restoring the entire backup irrespective of the configured memory and magnetic store retention periods. However, once the restore is completed, your restored table will only contain the data within the configured retention periods.

Restoring a Timestream for LiveAnalytics table to another Region or account

To restore a Timestream for LiveAnalytics table to another Region or account, you will first need to copy the backup to that new Region or account. In order to copy to another account, that account must first grant you permission. After you have copied your Timestream for LiveAnalytics backup to the new Region or account, it can be restored with the process in the previous section.

Copying a backup of a Amazon Timestream table

You can make a copy of a current backup. You can copy backups to multiple AWS accounts or AWS Regions on demand or automatically as part of a scheduled backup plan. Cross-Region replication is especially valuable if you have business continuity or compliance requirements to store backups a minimum distance away from your production data.

Cross-account backups are useful for securely copying your backups to one or more AWS accounts in your organization for operational or security reasons. If your original backup is inadvertently deleted, you can copy the backup from its destination account to its source account, and then start the restore. Before you can do this, you must have two accounts that belong to the same organization in the Organizations service and required permissions for the accounts. When you copy an incremental backup into another account or Region, the associated full backup is also copied.

Copies inherit the source backup's configuration unless you specify otherwise. There is one exception. If you specify your new copy to "Never" expire. With this setting, the new copy still inherits its source expiration date. If you want your new backup copy to be permanent, either set your source backups to never expire, or specify your new copy to expire 100 years after its creation.

To copy a backup from Timestream console, follow these steps.

1. Sign in to the AWS Management Console.
2. In the navigation pane on the left side of the console, choose **Backups**.

3. Choose the radio button next to the recovery point ID of the resource. In the upper-right corner of the pane, select **Actions** and choose **Copy**.
4. Select **Continue to AWS Backup** and follow the steps for Cross account backup.

Copying on-demand and scheduled backups across accounts and Regions is not natively supported in the Timestream for LiveAnalytics console currently and you have to navigate to AWS Backup to perform the operation.

Deleting backups

This section describes how to delete a backup of a Timestream for LiveAnalytics table.

To delete a backup from Timestream console, follow these steps.

1. Sign in to the AWS Management Console.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. Choose the radio button next to the recovery point ID of the resource. In the upper-right corner of the pane, select **Actions** and choose **Delete**.
4. Select **Continue to AWS Backup** and follow the steps for deleting backups at Deleting backups.

> **Note**
>
> When you delete a backup that is incremental, only the incremental backup is deleted and the underlying full backup is not deleted.

Quota and limits

AWS Backup limits the backups to one concurrent backup per resource. Therefore, additional scheduled or on-demand backup requests for the resource are queued and will start only after the existing backup job is completed. If the backup job is not started or completed within the backup window, the request fails. For more information about AWS Backup limits, see AWS Backup Limits in the AWS Backup Developer Guide.

When creating a backup, you can execute up to four concurrent backups per account. Similarly, you can execute one concurrent restore per account. When you initiate more than four backup

jobs simultaneously, only four backup jobs are initiated and the remaining jobs will be periodically retried. Once initiated, if the backup job is not completed within the configured backup window duration, the backup job fails. If the failed backup job is an on-demand backup, you can retry the backup and for scheduled backups, the job is attempted in the following schedule.

Customer-defined partition keys

Amazon Timestream for LiveAnalytics customer-defined partition keys is a feature in Timestream for LiveAnalytics that enables customers to define their own partition keys for their tables. Partitioning is a technique used to distribute data across multiple physical storage units, allowing for faster and more efficient data retrieval. With customer-defined partition keys, customers can create a partitioning schema that better aligns with their query patterns and use cases.

With Timestream for LiveAnalytics customer-defined partition keys, customers can choose one dimension names as a partition key for their tables. This allows for more flexibility in defining the partitioning schema for their data. By selecting the right partition key, customers can optimize their data model, improving their query performance, and reduce query latency.

Topics

- Using customer-defined partition keys
- Getting started with customer-defined partition keys
- Checking partitioning schema configuration
- Updating partitioning schema configuration
- Advantages of customer-defined partition keys
- Limitations of customer-defined partition keys
- Customer-defined partition keys and low cardinality dimensions
- Creating partition keys for existing tables
- Timestream for LiveAnalytics schema validation with custom composite partition keys

Using customer-defined partition keys

If you have a well-defined query pattern with high cardinality dimensions and require low query latency, a Timestream for LiveAnalytics customer-defined partition key can be a useful tool to enhance your data model. For instance, if you are a retail company tracking customer interactions on your website, the main access patterns would likely be by customer ID and timestamp. By

defining customer ID as the partition key, your data can be distributed evenly, allowing for reduced latency, ultimately improving the user experience.

Another example is in the healthcare industry, where wearable devices collect sensor data to track patients' vital signs. The main access pattern would be by Device ID and timestamp, with high cardinality on both dimensions. By defining Device ID as the partition key, can optimize your query execution and ensure a sustained long term query performance.

In summary, Timestream for LiveAnalytics customer-defined partition keys are most useful when you have a clear query pattern, high cardinality dimensions, and need low latency for your queries. By defining a partition key that aligns with your query pattern, you can optimize your query execution and ensure a sustained long term performance query performance.

Getting started with customer-defined partition keys

From the console, choose **Tables** and create a new table. You can also use an SDK to access the `CreateTable` action to create new tables that can include a customer-defined partition key.

Create a table with a dimension type partition key

You can use the following code snippets to create a table with a dimension type partition key.

Java

```
public void createTableWithDimensionTypePartitionKeyExample() {
        System.out.println("Creating table");
        CreateTableRequest createTableRequest = new CreateTableRequest();
        createTableRequest.setDatabaseName(DATABASE_NAME);
        createTableRequest.setTableName(TABLE_NAME);
        final RetentionProperties retentionProperties = new RetentionProperties()
                .withMemoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                .withMagneticStoreRetentionPeriodInDays(CT_TTL_DAYS);
        createTableRequest.setRetentionProperties(retentionProperties);

        // Can specify enforcement level with OPTIONAL or REQUIRED
        final List<PartitionKey> partitionKeyWithDimensionAndOptionalEnforcement =
 Collections.singletonList(new PartitionKey()
            .withName(COMPOSITE_PARTITION_KEY_DIM_NAME)
            .withType(PartitionKeyType.DIMENSION)
            .withEnforcementInRecord(PartitionKeyEnforcementLevel.OPTIONAL));
        Schema schema = new Schema();
```

```
 schema.setCompositePartitionKey(partitionKeyWithDimensionAndOptionalEnforcement);
        createTableRequest.setSchema(schema);

        try {
            writeClient.createTable(createTableRequest);
            System.out.println("Table [" + TABLE_NAME + "] successfully created.");
        } catch (ConflictException e) {
            System.out.println("Table [" + TABLE_NAME + "] exists on database [" +
 DATABASE_NAME + "] . Skipping database creation");
        }
    }
```

Java v2

```
public void createTableWithDimensionTypePartitionKeyExample() {
        System.out.println("Creating table");
        final RetentionProperties retentionProperties =
 RetentionProperties.builder()
                .memoryStoreRetentionPeriodInHours(HT_TTL_HOURS)
                .magneticStoreRetentionPeriodInDays(CT_TTL_DAYS)
                .build();
        // Can specify enforcement level with OPTIONAL or REQUIRED
        final List<PartitionKey> partitionKeyWithDimensionAndOptionalEnforcement =
 Collections.singletonList(PartitionKey
                .builder()
                .name(COMPOSITE_PARTITION_KEY_DIM_NAME)
                .type(PartitionKeyType.DIMENSION)
                .enforcementInRecord(PartitionKeyEnforcementLevel.OPTIONAL)
                .build());
        final Schema schema = Schema.builder()

 .compositePartitionKey(partitionKeyWithDimensionAndOptionalEnforcement).build();
        final CreateTableRequest createTableRequest = CreateTableRequest.builder()
                .databaseName(DATABASE_NAME)
                .tableName(TABLE_NAME)
                .retentionProperties(retentionProperties)
                .schema(schema)
                .build();

        try {
            writeClient.createTable(createTableRequest);
            System.out.println("Table [" + TABLE_NAME + "] successfully created.");
```

```
        } catch (ConflictException e) {
            System.out.println("Table [" + TABLE_NAME + "] exists on database [" +
 DATABASE_NAME + "] . Skipping database creation");
        }
    }
```

Go v1

```
func createTableWithDimensionTypePartitionKeyExample(){
        // Can specify enforcement level with OPTIONAL or REQUIRED
        partitionKeyWithDimensionAndOptionalEnforcement :=
 []*timestreamwrite.PartitionKey{
                {
                    Name:                aws.String(CompositePartitionKeyDimName),
                    EnforcementInRecord: aws.String("OPTIONAL"),
                    Type:                aws.String("DIMENSION"),
                },
        }
        createTableInput := &timestreamwrite.CreateTableInput{
             DatabaseName: aws.String(*databaseName),
             TableName:    aws.String(*tableName),
             // Enable MagneticStoreWrite for Table
             MagneticStoreWriteProperties:
 &timestreamwrite.MagneticStoreWriteProperties{
                 EnableMagneticStoreWrites: aws.Bool(true),
                 // Persist MagneticStoreWrite rejected records in S3
                 MagneticStoreRejectedDataLocation:
 &timestreamwrite.MagneticStoreRejectedDataLocation{
                     S3Configuration: &timestreamwrite.S3Configuration{
                         BucketName:       aws.String("timestream-sample-bucket"),
                         ObjectKeyPrefix:  aws.String("TimeStreamCustomerSampleGo"),
                         EncryptionOption: aws.String("SSE_S3"),
                     },
                 },
             },
             Schema: &timestreamwrite.Schema{
                 CompositePartitionKey:
 partitionKeyWithDimensionAndOptionalEnforcement,
             }
         }
         _, err := writeSvc.CreateTable(createTableInput)
    }
```

Go v2

```
 func (timestreamBuilder TimestreamBuilder)
CreateTableWithDimensionTypePartitionKeyExample() error {
        partitionKeyWithDimensionAndOptionalEnforcement := []types.PartitionKey{
            {
                Name:                aws.String(CompositePartitionKeyDimName),
                EnforcementInRecord: types.PartitionKeyEnforcementLevelOptional,
                Type:                types.PartitionKeyTypeDimension,
            },
        }
        _, err := timestreamBuilder.WriteSvc.CreateTable(context.TODO(),
&timestreamwrite.CreateTableInput{
            DatabaseName: aws.String(databaseName),
            TableName:    aws.String(tableName),
            MagneticStoreWriteProperties: &types.MagneticStoreWriteProperties{
                EnableMagneticStoreWrites: aws.Bool(true),
                // Persist MagneticStoreWrite rejected records in S3
                MagneticStoreRejectedDataLocation:
&types.MagneticStoreRejectedDataLocation{
                    S3Configuration: &types.S3Configuration{
                        BucketName:       aws.String(s3BucketName),
                        EncryptionOption: "SSE_S3",
                    },
                },
            },
            Schema: &types.Schema{
                CompositePartitionKey:
partitionKeyWithDimensionAndOptionalEnforcement,
            },
        })

        if err != nil {
            fmt.Println("Error:")
            fmt.Println(err)
        } else {
            fmt.Println("Create table is successful")
        }
        return err
    }
```

Python

```
def create_table_with_measure_name_type_partition_key(self):
        print("Creating table")
        retention_properties = {
            'MemoryStoreRetentionPeriodInHours': HT_TTL_HOURS,
            'MagneticStoreRetentionPeriodInDays': CT_TTL_DAYS
        }
        partitionKey_with_measure_name = [
            {'Type': 'MEASURE'}
        ]
        schema = {
            'CompositePartitionKey': partitionKey_with_measure_name
        }
        try:
            self.client.create_table(DatabaseName=DATABASE_NAME,
 TableName=TABLE_NAME,
                                     RetentionProperties=retention_properties,
 Schema=schema)
            print("Table [%s] successfully created." % TABLE_NAME)
        except self.client.exceptions.ConflictException:
            print("Table [%s] exists on database [%s]. Skipping table creation" % (
                TABLE_NAME, DATABASE_NAME))
        except Exception as err:
            print("Create table failed:", err)
```

Checking partitioning schema configuration

You can check how a table configuration for partitioning schema in a couple of ways. From the console, choose **Databases** and choose the table to check. You can also use an SDK to access the `DescribeTable` action.

Describe a table with a partition key

You can use the following code snippets to describe a table with a partition key.

Java

```
    public void describeTable() {
        System.out.println("Describing table");
        final DescribeTableRequest describeTableRequest = new
 DescribeTableRequest();
```

```
        describeTableRequest.setDatabaseName(DATABASE_NAME);
        describeTableRequest.setTableName(TABLE_NAME);
        try {
            DescribeTableResult result =
amazonTimestreamWrite.describeTable(describeTableRequest);
            String tableId = result.getTable().getArn();
            System.out.println("Table " + TABLE_NAME + " has id " + tableId);
            // If table is created with composite partition key, it can be described
with
            //
System.out.println(result.getTable().getSchema().getCompositePartitionKey());
        } catch (final Exception e) {
            System.out.println("Table " + TABLE_NAME + " doesn't exist = " + e);
            throw e;
        }
    }
```

The following is an example output.

1. Table has dimension type partition key

```
[{Type: DIMENSION,Name: hostId,EnforcementInRecord: OPTIONAL}]
```

2. Table has measure name type partition key

```
[{Type: MEASURE,}]
```

3. Getting composite partition key from a table created without specifying composite partition key

```
[{Type: MEASURE,}]
```

Java v2

```
 public void describeTable() {
        System.out.println("Describing table");
        final DescribeTableRequest describeTableRequest =
DescribeTableRequest.builder()
                .databaseName(DATABASE_NAME).tableName(TABLE_NAME).build();
        try {
```

```
            DescribeTableResponse response =
writeClient.describeTable(describeTableRequest);
            String tableId = response.table().arn();
            System.out.println("Table " + TABLE_NAME + " has id " + tableId);
            // If table is created with composite partition key, it can be described
with
            //
System.out.println(response.table().schema().compositePartitionKey());
        } catch (final Exception e) {
            System.out.println("Table " + TABLE_NAME + " doesn't exist = " + e);
            throw e;
        }
    }
```

The following is an example output.

1. Table has dimension type partition key

```
[PartitionKey(Type=DIMENSION, Name=hostId, EnforcementInRecord=OPTIONAL)]
```

2. Table has measure name type partition key

```
[PartitionKey(Type=MEASURE)]
```

3. Getting composite partition key from a table created without specifying composite partition key will return

```
[PartitionKey(Type=MEASURE)]
```

Go v1

```
    <tablistentry>
     <tabname> Go </tabname>
     <tabcontent>
      <programlisting language="go"></programlisting>
     </tabcontent>
    </tablistentry>
```

The following is an example output.

```
{
```

```
  Table: {
    Arn: "arn:aws:timestream:us-west-2:533139590831:database/devops/table/
host_metrics_dim_pk_1",
    CreationTime: 2023-05-31 01:52:00.511 +0000 UTC,
    DatabaseName: "devops",
    LastUpdatedTime: 2023-05-31 01:52:00.511 +0000 UTC,
    MagneticStoreWriteProperties: {
      EnableMagneticStoreWrites: true,
      MagneticStoreRejectedDataLocation: {
        S3Configuration: {
          BucketName: "timestream-sample-bucket-west",
          EncryptionOption: "SSE_S3",
          ObjectKeyPrefix: "TimeStreamCustomerSampleGo"
        }
      }
    },
    RetentionProperties: {
      MagneticStoreRetentionPeriodInDays: 73000,
      MemoryStoreRetentionPeriodInHours: 6
    },
    Schema: {
      CompositePartitionKey: [{
          EnforcementInRecord: "OPTIONAL",
          Name: "hostId",
          Type: "DIMENSION"
        }]
    },
    TableName: "host_metrics_dim_pk_1",
    TableStatus: "ACTIVE"
  }
}
```

Go v2

```
 func (timestreamBuilder TimestreamBuilder) DescribeTable()
 (*timestreamwrite.DescribeTableOutput, error) {
        describeTableInput := &timestreamwrite.DescribeTableInput{
            DatabaseName: aws.String(databaseName),
            TableName:    aws.String(tableName),
        }
       describeTableOutput, err :=
 timestreamBuilder.WriteSvc.DescribeTable(context.TODO(), describeTableInput)
```

```
        if err != nil {
            fmt.Printf("Failed to describe table with Error: %s", err.Error())
        } else {
            fmt.Printf("Describe table is successful : %s\n",
 JsonMarshalIgnoreError(*describeTableOutput))
            // If table is created with composite partition key, it will be included
 in the output
        }

        return describeTableOutput, err
    }
```

The following is an example output.

```
{
  "Table": {
    "Arn":"arn:aws:timestream:us-east-1:351861611069:database/cdpk-wr-db/table/
host_metrics_dim_pk",
    "CreationTime":"2023-05-31T22:36:10.66Z",
    "DatabaseName":"cdpk-wr-db",
    "LastUpdatedTime":"2023-05-31T22:36:10.66Z",
    "MagneticStoreWriteProperties":{
      "EnableMagneticStoreWrites":true,
      "MagneticStoreRejectedDataLocation":{
        "S3Configuration":{
          "BucketName":"error-configuration-sample-s3-bucket-cq8my",
          "EncryptionOption":"SSE_S3",
          "KmsKeyId":null,"ObjectKeyPrefix":null
        }
      }
    },
    "RetentionProperties":{
      "MagneticStoreRetentionPeriodInDays":73000,
      "MemoryStoreRetentionPeriodInHours":6
    },
    "Schema":{
      "CompositePartitionKey":[{
        "Type":"DIMENSION",
        "EnforcementInRecord":"OPTIONAL",
        "Name":"hostId"
      }]
    },
    "TableName":"host_metrics_dim_pk",
```

```
    "TableStatus":"ACTIVE"
  },
  "ResultMetadata":{}
}
```

Python

```
 def describe_table(self):
       print('Describing table')
       try:
           result = self.client.describe_table(DatabaseName=DATABASE_NAME,
TableName=TABLE_NAME)
           print("Table [%s] has id [%s]" % (TABLE_NAME, result['Table']['Arn']))
           # If table is created with composite partition key, it can be described
with
           # print(result['Table']['Schema'])
       except self.client.exceptions.ResourceNotFoundException:
           print("Table doesn't exist")
       except Exception as err:
           print("Describe table failed:", err)
```

The following is an example output.

1. Table has dimension type partition key

```
[{'CompositePartitionKey': [{'Type': 'DIMENSION', 'Name': 'hostId',
 'EnforcementInRecord': 'OPTIONAL'}]}]
```

2. Table has measure name type partition key

```
[{'CompositePartitionKey': [{'Type': 'MEASURE'}]}]
```

3. Getting composite partition key from a table created without specifying composite partition key

```
[{'CompositePartitionKey': [{'Type': 'MEASURE'}]}]
```

Updating partitioning schema configuration

You can update table configuration for partitioning schema with an SDK with access the `UpdateTable` action.

Update a table with a partition key

You can use the following code snippets to update a table with a partition key.

Java

```
   public void updateTableCompositePartitionKeyEnforcement() {
       System.out.println("Updating table");

       UpdateTableRequest updateTableRequest = new UpdateTableRequest();
       updateTableRequest.setDatabaseName(DATABASE_NAME);
       updateTableRequest.setTableName(TABLE_NAME);

       // Can update enforcement level for dimension type partition key with
OPTIONAL or REQUIRED enforcement
       final List<PartitionKey> partitionKeyWithDimensionAndRequiredEnforcement =
Collections.singletonList(new PartitionKey()
           .withName(COMPOSITE_PARTITION_KEY_DIM_NAME)
           .withType(PartitionKeyType.DIMENSION)
           .withEnforcementInRecord(PartitionKeyEnforcementLevel.REQUIRED));
       Schema schema = new Schema();

schema.setCompositePartitionKey(partitionKeyWithDimensionAndRequiredEnforcement);
       updateTableRequest.withSchema(schema);

       writeClient.updateTable(updateTableRequest);
       System.out.println("Table updated");
```

Java v2

```
   public void updateTableCompositePartitionKeyEnforcement() {
       System.out.println("Updating table");
       // Can update enforcement level for dimension type partition key with
OPTIONAL or REQUIRED enforcement
       final List<PartitionKey> partitionKeyWithDimensionAndRequiredEnforcement =
Collections.singletonList(PartitionKey
           .builder()
           .name(COMPOSITE_PARTITION_KEY_DIM_NAME)
```

```
            .type(PartitionKeyType.DIMENSION)
            .enforcementInRecord(PartitionKeyEnforcementLevel.REQUIRED)
            .build());
        final Schema schema = Schema.builder()

.compositePartitionKey(partitionKeyWithDimensionAndRequiredEnforcement).build();
        final UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()

.databaseName(DATABASE_NAME).tableName(TABLE_NAME).schema(schema).build();

        writeClient.updateTable(updateTableRequest);
        System.out.println("Table updated");
```

Go v1

```
// Update table partition key enforcement attribute
   updateTableInput := &timestreamwrite.UpdateTableInput{
        DatabaseName: aws.String(*databaseName),
        TableName:    aws.String(*tableName),
        // Can update enforcement level for dimension type partition key with
OPTIONAL or REQUIRED enforcement
        Schema: &timestreamwrite.Schema{
            CompositePartitionKey: []*timestreamwrite.PartitionKey{
                {
                        Name:
aws.String(CompositePartitionKeyDimName),
                        EnforcementInRecord: aws.String("REQUIRED"),
                        Type:                aws.String("DIMENSION"),
                },
            }},
    }
    updateTableOutput, err := writeSvc.UpdateTable(updateTableInput)
        if err != nil {
            fmt.Println("Error:")
            fmt.Println(err)
        } else {
            fmt.Println("Update table is successful, below is the output:")
            fmt.Println(updateTableOutput)
        }
```

Go v2

```
// Update table partition key enforcement attribute
```

```
        updateTableInput := &timestreamwrite.UpdateTableInput{
            DatabaseName: aws.String(*databaseName),
            TableName:    aws.String(*tableName),
            // Can update enforcement level for dimension type partition key with
OPTIONAL or REQUIRED enforcement
            Schema: &types.Schema{
                CompositePartitionKey: []types.PartitionKey{
                    {
                        Name:
aws.String(CompositePartitionKeyDimName),
                        EnforcementInRecord:
types.PartitionKeyEnforcementLevelRequired,
                        Type:               types.PartitionKeyTypeDimension,
                    },
                }},
        }
        updateTableOutput, err :=
timestreamBuilder.WriteSvc.UpdateTable(context.TODO(), updateTableInput)
        if err != nil {
            fmt.Println("Error:")
            fmt.Println(err)
        } else {
            fmt.Println("Update table is successful, below is the output:")
            fmt.Println(updateTableOutput)
        }
```

Python

```
   def update_table(self):
       print('Updating table')
       try:
           # Can update enforcement level for dimension type partition key with
OPTIONAL or REQUIRED enforcement
           partition_key_with_dimension_and_required_enforcement = [
               {
                   'Type': 'DIMENSION',
                   'Name': COMPOSITE_PARTITION_KEY_DIM_NAME,
                   'EnforcementInRecord': 'REQUIRED'
               }
           ]
           schema = {
               'CompositePartitionKey':
partition_key_with_dimension_and_required_enforcement
```

```
            }
            self.client.update_table(DatabaseName=DATABASE_NAME,
TableName=TABLE_NAME,
                                     Schema=schema)
            print('Table updated.')
        except Exception as err:
            print('Update table failed:', err)
```

Advantages of customer-defined partition keys

Enhanced query performance: Customer-defined partition keys enable you to optimize your query execution and improve overall query performance. By defining partition keys that align with your query patterns, you can minimize data scanning and optimize data pruning, resulting in lower query latency.

Better long term performance predictability: Customer-defined partition keys allow customers to distribute data evenly across partitions, improving the efficiency of data management. This will ensure that your query performance remains stable as your data stored scales over time.

Limitations of customer-defined partition keys

As a Timestream for LiveAnalytics user, it's important to keep in mind the limitations around a customer partition key. Firstly, it requires a good understanding of your workload and query patterns. This means that you should have a clear idea of which dimensions are most frequently use as main filtering conditions in queries and have high cardinality to make the most effective use of partition keys.

Secondly, partition keys need to be defined at the time of table creation and cannot be added to existing tables. This means that you should carefully consider your partitioning strategy before creating a table to ensure that it aligns with your business needs.

Lastly, it's important to note that once the table has been created, you cannot change the partition key afterwards. This means that you should thoroughly test and evaluate your partitioning strategy before committing to it. With these limitations in mind, Timestream's customer-defined partition key can greatly improve query performance and long term satisfaction.

Customer-defined partition keys and low cardinality dimensions

If you decide to use a partition key with very low cardinality, such as a specific region or state, it is important to note that the data for for other entities such as `customerID`, `ProductCategory`,

and others, could end up spread across too many partitions sometimes with little or no data present. This can lead to inefficient query execution and decreased performance.

To avoid this, we recommend you choose dimensions that are not only part of your key filtering condition but have higher cardinality. This will help ensure that the data is evenly distributed across the partitions and improve query performance.

Creating partition keys for existing tables

If you already have tables in Timestream for LiveAnalytics and want to use customer-defined partition keys, you will need to migrate your data into a new table with the desired partitioning schema definition. This can be done using export to S3 and batch load together, which involves exporting the data from the existing table to S3, modifying the data to include the partition key (if necessary) and adding headers to your CSV files, and then importing the data into a new table with the desired partitioning schema defined. Keep in mind that this method can be time consuming and costly, especially for large tables.

Alternatively, you can use scheduled queries to migrate your data to a new table with the desired partitioning schema. This method involves creating a scheduled query that reads from the existing table and writes to the new table. The scheduled query can be set up to run on a regular basis until all the data has been migrated. Keep in mind that you will be charged for reading and writing the data during the migration process.

Timestream for LiveAnalytics schema validation with custom composite partition keys

Schema validation in Timestream for LiveAnalytics helps ensure that data ingested into the database complies with the specified schema, minimizing ingestion errors and improving data quality. In particular, schema validation is especially useful when adopting customer-defined partition key with the goal of optimizing your query performance.

What is Timestream for LiveAnalytics schema validation with customer-defined partition keys?

Timestream for LiveAnalytics schema validation is a feature that validates data being ingested into a Timestream for LiveAnalytics table based on a predefined schema. This schema defines the data model, including partition key, data types, and constraints for the records being inserted.

When using a customer-defined partition key, schema validation becomes even more crucial. Partition keys allow you to specify a partition key, which determines how your data is stored in Timestream for LiveAnalytics. By validating the incoming data against the schema with a custom partition key, you can enforce data consistency, detect errors early, and improve the overall quality of the data stored in Timestream for LiveAnalytics.

How to Use Timestream for LiveAnalytics schema validation with custom composite partition keys

To use Timestream for LiveAnalytics schema validation with custom composite partition keys, follow these steps:

Think about what your query patterns will look like: To properly choose and define the schema for your Timestream for LiveAnalytics table you should start with your query requirements.

Specify custom composite partition keys: When creating the table, specify a custom partition key. This key determines the attribute that will be used to partition the table data. You can choose between dimension keys and measure keys for partitioning. A dimension key partitions data based on a dimension name, while a measure key partitions data based on the measure name.

Set enforcement levels: To ensure proper data partitioning and the benefits that come with it, Amazon Timestream for LiveAnalytics allows you to set enforcement levels for each partition key in your schema. The enforcement level determines whether the partition key dimension is required or optional when ingesting records. You can choose between two options: `REQUIRED`, which means the partition key must be present in the ingested record, and `OPTIONAL`, which means the partition key doesn't have to be present. It is recommended that you use the `REQUIRED` enforcement level when using a customer-defined partition to ensure that your data is properly partitioned and you get the full benefits of this feature. Additionally, you can change the enforcement level configuration at any time after the schema creation to adjust to your data ingestion requirements.

Ingest data: When ingesting data into the Timestream for LiveAnalytics table, the schema validation process will check the records against the defined schema with custom composite partition keys. If the records do not adhere to the schema, Timestream for LiveAnalytics will return a validation error.

Handle validation errors: In case of validation errors, Timestream for LiveAnalytics will return a `ValidationException` or a `RejectedRecordsException`, depending on the type of error. Make sure to handle these exceptions in your application and take appropriate action, such as fixing the incorrect records and retrying the ingestion.

Update enforcement levels: If necessary, you can update the enforcement level of partition keys after table creation using the `UpdateTable` action. However, it's important to note that some aspects of the partition key configuration, such as the name, and type, cannot be changed after table creation. If you change the enforcement level from `REQUIRED` to `OPTIONAL`, all records will be accepted regardless of the presence of the attribute selected as the customer-defined partition key. Conversely, if you change the enforcement level from `OPTIONAL` to `REQUIRED`, you may start seeing 4xx write errors for records that don't meet this condition. Therefore, it's essential to choose the appropriate enforcement level for your use case when creating your table, based on your data's partitioning requirements.

When to use Timestream for LiveAnalytics schema validation with custom composite partition keys

Timestream for LiveAnalytics schema validation with custom composite partition keys should be used in scenarios where data consistency, quality, and optimized partitioning are crucial. By enforcing a schema during data ingestion, you can prevent errors and inconsistencies that might lead to incorrect analysis or loss of valuable insights.

Interaction with batch load jobs

When setting up a batch load job to import data into a table with a customer-defined partition key, there are a few scenarios that could affect the process:

1. If the enforcement level is set to `OPTIONAL`, an alert will be displayed on the console during the creation flow if the partition key is not mapped during job configuration. This alert will not appear when using the API or CLI.
2. If the enforcement level is set to `REQUIRED`, the job creation will be rejected unless the partition key is mapped to a source data column.
3. If the enforcement level is changed to `REQUIRED` after the job is created, the job will continue to execute, but any records that do not have the proper mapping for the partition key will be rejected with a 4xx error.

Interaction with scheduled query

When setting up a scheduled query job for calculating and storing aggregates, rollups, and other forms of preprocessed data into a table with a customer-defined partition key, there are a few scenarios that could affect the process:

1. If the enforcement level is set to `OPTIONAL`, an alert will be displayed if the partition key is not mapped during job configuration. This alert will not appear when using the API or CLI.
2. If the enforcement level is set to `REQUIRED`, the job creation will be rejected unless the partition key is mapped to a source data column.
3. If the enforcement level is changed to `REQUIRED` after the job is created and the scheduled query results does not contain the partition key dimension, all the next iterations of the job will fail.

Adding tags and labels to resources

You can label Amazon Timestream for LiveAnalytics resources using *tags*. Tags let you categorize your resources in different ways—for example, by purpose, owner, environment, or other criteria. Tags can help you do the following:

- Quickly identify a resource based on the tags that you assigned to it.
- See AWS bills broken down by tags.

Tagging is supported by AWS services like Amazon Elastic Compute Cloud (Amazon EC2), Amazon Simple Storage Service (Amazon S3), Timestream for LiveAnalytics, and more. Efficient tagging can provide cost insights by enabling you to create reports across services that carry a specific tag.

To get started with tagging, do the following:

1. Understand Tagging restrictions.
2. Create tags by using Tagging operations.

Finally, it is good practice to follow optimal tagging strategies. For information, see AWS Tagging Strategies.

Tagging restrictions

Each tag consists of a key and a value, both of which you define. The following restrictions apply:

- Each Timestream for LiveAnalytics table can have only one tag with the same key. If you try to add an existing tag, the existing tag value is updated to the new value.

- A value acts as a descriptor within a tag category. In Timestream for LiveAnalytics the value cannot be empty or null.
- Tag keys and values are case sensitive.
- The maximum key length is 128 Unicode characters.
- The maximum value length is 256 Unicode characters.
- The allowed characters are letters, white space, and numbers, plus the following special characters: `+ - = . _ : /`
- The maximum number of tags per resource is 50.
- AWS-assigned tag names and values are automatically assigned the `aws:` prefix, which you can't assign. AWS-assigned tag names don't count toward the tag limit of 50. User-assigned tag names have the prefix `user:` in the cost allocation report.
- You can't backdate the application of a tag.

Tagging operations

You can add, list, edit, or delete tags for databases and tables using the Amazon Timestream for LiveAnalytics console, query language, or the AWS Command Line Interface (AWS CLI).

Topics

- Adding tags to new or existing databases and tables using the console

Adding tags to new or existing databases and tables using the console

You can use the Timestream for LiveAnalytics console to add tags to new databases, tables and scheduled queries when you create them. You can also add, edit, or delete tags for existing tables.

To tag databases when creating them (console)

1. Open the Timestream console at https://console.aws.amazon.com/timestream.
2. In the navigation pane, choose **Databases**, and then choose **Create database**.
3. On the **Create database** page, provide a name for the database. Enter a key and value for the tag, and then choose **Add new tag**.
4. Choose **Create database**.

To tag tables when creating them (console)

1. Open the Timestream console at https://console.aws.amazon.com/timestream.
2. In the navigation pane, choose **Tables**, and then choose **Create table**.
3. On the **Create Timestream for LiveAnalytics table** page, provide a name for the table. Enter a key and value for the tag, and choose **Add new tag**.
4. Choose **Create table**.

To tag scheduled queries when creating them (console)

1. Open the Timestream console at https://console.aws.amazon.com/timestream.
2. In the navigation pane, choose **Scheduled queries**, and then choose **Create scheduled query**.
3. On the **Step 3. Configure query settings** page, choose **Add new tag**. Enter a key and value for the tag. Choose **Add new tag** to add additional tags.
4. Choose **Next**.

To tag existing resources (console)

1. Open the Timestream console at https://console.aws.amazon.com/timestream.
2. In the navigation pane, choose **Databases**, **Tables** or **Scheduled queries**.
3. Choose a database or table in the list. Then choose **Manage tags** to add, edit, or delete your tags.

For information about tag structure, see Tagging restrictions.

Security in Timestream for LiveAnalytics

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The

effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to Timestream for LiveAnalytics, see AWS Services in Scope by Compliance Program.

- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using Timestream for LiveAnalytics. The following topics show you how to configure Timestream for LiveAnalytics to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your Timestream for LiveAnalytics resources.

Topics

- Data protection in Timestream for LiveAnalytics
- Identity and access management for Amazon Timestream for LiveAnalytics
- Logging and monitoring in Timestream for LiveAnalytics
- Resilience in Amazon Timestream Live Analytics
- Infrastructure security in Amazon Timestream Live Analytics
- Configuration and vulnerability analysis in Timestream
- Incident response in Timestream for LiveAnalytics
- VPC endpoints (AWS PrivateLink)
- Security best practices for Amazon Timestream for LiveAnalytics

Data protection in Timestream for LiveAnalytics

The AWS shared responsibility model applies to data protection in Amazon Timestream Live Analytics. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Timestream Live Analytics or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

For more detailed information on Timestream for LiveAnalytics data protection topics like Encryption at Rest and Key Management, select any of the available topics below.

Topics

- Encryption at rest
- Encryption in transit
- Key management

Encryption at rest

Timestream for LiveAnalytics encryption at rest provides enhanced security by encrypting all your data at rest using encryption keys stored in AWS Key Management Service (AWS KMS). This functionality helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet strict encryption compliance and regulatory requirements.

- Encryption is turned on by default on your Timestream for LiveAnalytics database, and cannot be turned off. The industry standard AES-256 encryption algorithm is the default encryption algorithm used.
- AWS KMS is required for encryption at rest in Timestream for LiveAnalytics.
- You cannot encrypt only a subset of items in a table.
- You don't need to modify your database client applications to use encryption.

If you do not provide a key, Timestream for LiveAnalytics creates and uses an AWS KMS key named `alias/aws/timestream` in your account.

You may use your own customer managed key in KMS to encrypt your Timestream for LiveAnalytics data. For more information on keys in Timestream for LiveAnalytics, see Key management.

Timestream for LiveAnalytics stores your data in two storage tiers, memory store and magnetic store. Memory store data is encrypted using a Timestream for LiveAnalytics service key. Magnetic store data is encrypted using your AWS KMS key.

The Timestream Query service requires credentials to access your data. These credentials are encrypted using your KMS key.

> **Note**
>
> Timestream for LiveAnalytics doesn't call AWS KMS for every Decrypt operation. Instead, it maintains a local cache of keys for 5 minutes with active traffic. Any permission changes are propagated through the Timestream for LiveAnalytics system with eventual consistency within at most 5 minutes.

Encryption in transit

All your Timestream Live Analytics data is encrypted in transit. By default, all communications to and from Timestream for LiveAnalytics are protected by using Transport Layer Security (TLS) encryption.

Key management

You can manage keys for Amazon Timestream Live Analytics using the AWS Key Management Service (AWS KMS). **Timestream Live Analytics requires the use of KMS to encrypt your data.** You have the following options for key management, depending on how much control you require over your keys:

Database and table resources

- *Timestream Live Analytics-managed key:* If you do not provide a key, Timestream Live Analytics will create a `alias/aws/timestream` key using KMS.
- *Customer managed key:* KMS customer managed keys are supported. Choose this option if you require more control over the permissions and lifecycle of your keys, including the ability to have them automatically rotated on an annual basis.

Scheduled query resource

- *Timestream Live Analytics-owned key:* If you do not provide a key, Timestream Live Analytics will use its own a KMS key to encrypt the Query resource, this key is present in timestream account. See AWS owned keys in the KMS developer guide for more details.
- *Customer managed key:* KMS customer managed keys are supported. Choose this option if you require more control over the permissions and lifecycle of your keys, including the ability to have them automatically rotated on an annual basis.

KMS keys in an external key store (XKS) are not supported.

Identity and access management for Amazon Timestream for LiveAnalytics

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in)

and *authorized* (have permissions) to use Timestream for LiveAnalytics resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon Timestream for LiveAnalytics works with IAM
- AWS managed policies for Amazon Timestream Live Analytics
- Amazon Timestream for LiveAnalytics identity-based policy examples
- Troubleshooting Amazon Timestream for LiveAnalytics identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting Amazon Timestream for LiveAnalytics identity and access)
- **Service administrator** - determine user access and submit permission requests (see How Amazon Timestream for LiveAnalytics works with IAM)
- **IAM administrator** - write policies to manage access (see Amazon Timestream for LiveAnalytics identity-based policy examples)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.

- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Timestream for LiveAnalytics works with IAM

Before you use IAM to manage access to Timestream for LiveAnalytics, you should understand what IAM features are available to use with Timestream for LiveAnalytics. To get a high-level view of how Timestream for LiveAnalytics and other AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Topics

- Timestream for LiveAnalytics identity-based policies
- Timestream for LiveAnalytics resource-based policies
- Authorization based on Timestream for LiveAnalytics tags
- Timestream for LiveAnalytics IAM roles

Timestream for LiveAnalytics identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. Timestream for LiveAnalytics supports specific actions and resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

You can specify the following actions in the Action element of an IAM policy statement. Use policies to grant permissions to perform an operation in AWS. When you use an action in a policy, you usually allow or deny access to the API operation, CLI command or SQL command with the same name.

In some cases, a single action controls access to an API operation as well as SQL command. Alternatively, some operations require several different actions.

For a list of supported Timestream for LiveAnalytics `Action`'s, see the table below:

> **Note**
>
> For all database-specific `Actions`, you can specify a database ARN to limit the action to a particular database.

Actions	Description	Access level	Resource types (*required)
DescribeEndpoints	Returns the Timestream endpoint that subsequent requests must be made to.	All	*
Select	Run queries on Timestream that select data from one or more tables. See this note for a detailed explanation	Read	table*
CancelQuery	Cancel a query.	Read	*
ListTables	Get the list of tables.	List	database*
ListDatabases	Get the list of databases.	List	*

Actions	Description	Access level	Resource types (*required)
ListMeasures	Get the list of measures.	Read	table*
DescribeTable	Get the table description.	Read	table*
DescribeDatabase	Get the database description.	Read	database*
SelectValues	Run queries that do not require a particular resource to be specified. See this note for a detailed explanation.	Read	*
WriteRecords	Insert data into Timestream.	Write	table*
CreateTable	Create a table.	Write	database*
CreateDatabase	Create a database.	Write	*
DeleteDatabase	Delete a database.	Write	*
UpdateDatabase	Update a database.	Write	*
DeleteTable	Delete a table.	Write	database*
UpdateTable	Update a table.	Write	database*

SelectValues vs. select:

`SelectValues` is an `Action` that is used for queries that *do not* require a resource. An example of a query that does not require a resource is as follows:

```
SELECT 1
```

Notice that this query does not refer to a particular Timestream for LiveAnalytics resource. Consider another example:

```
SELECT now()
```

This query returns the current timestamp using the `now()` function, but does not require a resource to be specified. `SelectValues` is often used for testing, so that Timestream for LiveAnalytics can run queries without resources. Now, consider a `Select` query:

```
SELECT * FROM database.table
```

This type of query requires a resource, specifcially an Timestream for LiveAnalytics `table` , so that the specified data can be fetched from the table.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

In Timestream for LiveAnalytics databases and tables can be used in the `Resource` element of IAM permissions.

The Timestream for LiveAnalytics database resource has the following ARN:

```
arn:${Partition}:timestream:${Region}:${Account}:database/${DatabaseName}
```

The Timestream for LiveAnalytics table resource has the following ARN:

```
arn:${Partition}:timestream:${Region}:${Account}:database/${DatabaseName}/table/
${TableName}
```

For more information about the format of ARNs, see Amazon Resource Names (ARNs) and AWS Service Namespaces.

For example, to specify the `database` keyspace in your statement, use the following ARN:

```
"Resource": "arn:aws:timestream:us-east-1:123456789012:database/mydatabase"
```

To specify all databases that belong to a specific account, use the wildcard (*):

```
"Resource": "arn:aws:timestream:us-east-1:123456789012:database/*"
```

Some Timestream for LiveAnalytics actions, such as those for creating resources, cannot be performed on a specific resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

Condition keys

Timestream for LiveAnalytics does not provide any service-specific condition keys, but it does support using some global condition keys. To see all AWS global condition keys, see AWS Global Condition Context Keys in the *IAM User Guide*.

Examples

To view examples of Timestream for LiveAnalytics identity-based policies, see Amazon Timestream for LiveAnalytics identity-based policy examples.

Timestream for LiveAnalytics resource-based policies

Timestream for LiveAnalytics does not support resource-based policies. To view an example of a detailed resource-based policy page, see https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based.html.

Authorization based on Timestream for LiveAnalytics tags

You can manage access to your Timestream for LiveAnalytics resources by using tags. To manage resource access based on tags, you provide tag information in the condition element of a policy using the `timestream:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys. For more information about tagging Timestream for LiveAnalytics resources, see the section called “Tagging resources”.

To view example identity-based policies for limiting access to a resource based on the tags on that resource, see Timestream for LiveAnalytics resource access based on tags.

Timestream for LiveAnalytics IAM roles

An IAM role is an entity within your AWS account that has specific permissions.

Using temporary credentials with Timestream for LiveAnalytics

You can use temporary credentials to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS STS API operations such as AssumeRole or GetFederationToken.

Service-linked roles

Timestream for LiveAnalytics does not support service-linked roles.

Service roles

Timestream for LiveAnalytics does not support service roles.

AWS managed policies for Amazon Timestream Live Analytics

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

Topics

- AWS managed policy: AmazonTimestreamInfluxDBFullAccess
- AWS managed policy: AmazonTimestreamReadOnlyAccess
- AWS managed policy: AmazonTimestreamConsoleFullAccess
- AWS managed policy: AmazonTimestreamFullAccess
- Timestream Live Analytics updates to AWS managed policies

AWS managed policy: AmazonTimestreamInfluxDBFullAccess

You can attach `AmazonTimestreamInfluxDBFullAccess` to your users, groups, and roles. The policy access to create, update, delete and list Amazon Timestream InfluxDB instances.

Permission details

This policy includes the following permission:

- `Amazon Timestream` – Provides full administrative access to create, update, delete and list Amazon Timestream InfluxDB instances and create and list parameter groups.

To review this policy in JSON format, see AmazonTimestreamInfluxDBFullAccess.

AWS managed policy: AmazonTimestreamReadOnlyAccess

You can attach `AmazonTimestreamReadOnlyAccess` to your users, groups, and roles. The policy provides read-only access to Amazon Timestream.

Permission details

This policy includes the following permission:

- `Amazon Timestream` – Provides read-only access to Amazon Timestream. This policy also grants permission to cancel any running query.

To review this policy in JSON format, see AmazonTimestreamReadOnlyAccess.

AWS managed policy: AmazonTimestreamConsoleFullAccess

You can attach `AmazonTimestreamConsoleFullAccess` to your users, groups, and roles.

The policy provides full access to manage Amazon Timestream using the AWS Management Console. This policy also grants permissions for certain AWS KMS operations and operations to manage your saved queries.

Permission details

This policy includes the following permissions:

- `Amazon Timestream` – Grants principals full access to Amazon Timestream.
- `AWS KMS` – Allows principals to list aliases and describe keys.
- `Amazon S3` – Allows principals to list all Amazon S3 buckets.
- `Amazon SNS` – Allows principals to list Amazon SNS topics.
- `IAM` – Allows principals to list IAM roles.
- `DBQMS` – Allows principals to access, create, delete, describe, and update queries. The Database Query Metadata Service (dbqms) is an internal-only service. It provides your recent and saved queries for the query editor on the AWS Management Console for multiple AWS services, including Amazon Timestream.
- `Pricing` – Allows principals to access pricing estimation for InfluxDB resource configuration during creation.
- `Marketplace` – Allows principals to access marketplace resources and create agreements for InfluxDB Cluster with Read Replicas creation.

To review this policy in JSON format, see AmazonTimestreamConsoleFullAccess.

AWS managed policy: AmazonTimestreamFullAccess

You can attach `AmazonTimestreamFullAccess` to your users, groups, and roles.

The policy provides full access to Amazon Timestream. This policy also grants permissions for certain AWS KMS operations.

Permission details

This policy includes the following permissions:

- `Amazon Timestream` – Grants principals full access to Amazon Timestream.

- `AWS KMS` – Allows principals to list aliases and describe keys.
- `Amazon S3` – Allows principals to list all Amazon S3 buckets.

To review this policy in JSON format, see AmazonTimestreamFullAccess.

Timestream Live Analytics updates to AWS managed policies

View details about updates to AWS managed policies for Timestream Live Analytics since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Timestream Live Analytics Document history page.

Change	Description	Date
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Timestream for InfluxDB has added Influx Enterprise marketplace product ID to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy to support subscription to enterprise marketplace offerings. These permissions are restricted to specific AWS Marketplace products through a condition that limits the access to only certain `ProductIds`. See AmazonTimestreamInfluxDBFullAccess.	October 17, 2025
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Added the AWS Marketplace permissions to the existing `AmazonTimestreamConsoleFullAccess` managed policy to access marketplace resources	August 20, 2025

Change	Description	Date
	and create agreements for InfluxDB Cluster with Read Replicas creation. Timestream Live Analytics has also updated this managed policy by adding an `Sid` field. The policy update doesn't impact the usage of the `AmazonTimestreamConsoleFullAccess` managed policy.	
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Added the `pricing:GetProducts` action to the existing `AmazonTimestreamConsoleFullAccess` managed policy to provide pricing estimations for InfluxDB resource configurations during creation. The policy update doesn't impact the usage of the `AmazonTimestreamConsoleFullAccess` managed policy.	June 10, 2025

Change	Description	Date
AmazonTimestreamReadOnlyAccess – Update to an existing policy	Added the `timestream:DescribeAccountSettings` action to the existing `AmazonTimestreamReadOnlyAccess` managed policy. This action is used for describing AWS account settings. Timestream Live Analytics has also updated this managed policy by adding an `Sid` field. The policy update doesn't impact the usage of the `AmazonTimestreamReadOnlyAccess` managed policy.	June 03, 2024
AmazonTimestreamReadOnlyAccess – Update to an existing policy	Added the `timestream:DescribeBatchLoadTask` and `timestream:ListBatchLoadTasks` actions to the existing `AmazonTimestreamReadOnlyAccess` managed policy. These actions are used when listing and describing batch load tasks. The policy update doesn't impact the usage of the `AmazonTimestreamReadOnlyAccess` managed policy.	February 24, 2023

Change	Description	Date
AmazonTimestreamReadOnlyAccess – Update to an existing policy	Added the `timestream:DescribeScheduledQuery` and `timestream:ListScheduledQueries` actions to the existing `AmazonTimestreamReadOnlyAccess` managed policy. These actions are used when listing and describing existing scheduled queries. The policy update doesn't impact the usage of the `AmazonTimestreamReadOnlyAccess` managed policy.	November 29, 2021
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Added the `s3:ListAllMyBuckets` action to the existing `AmazonTimestreamConsoleFullAccess` managed policy. This action is used when you specify an Amazon S3 bucket for Timestream to log magnetic store write errors. The policy update doesn't impact the usage of the `AmazonTimestreamConsoleFullAccess` managed policy.	November 29, 2021

Change	Description	Date
AmazonTimestreamFullAccess – Update to an existing policy	Added the `s3:ListAllMyBuckets` action to the existing `AmazonTimestreamFullAccess` managed policy. This action is used when you specify an Amazon S3 bucket for Timestream to log magnetic store write errors. The policy update doesn't impact the usage of the `AmazonTimestreamFullAccess` managed policy.	November 29, 2021
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Removed redundant actions from the existing `AmazonTimestreamConsoleFullAccess` managed policy. Previously, this policy included a redundant action `dbqms:DescribeQueryHistory`. The updated policy removes the redundant action. The policy update doesn't impact the usage of the `AmazonTimestreamConsoleFullAccess` managed policy.	April 23, 2021
Timestream Live Analytics started tracking changes	Timestream Live Analytics started tracking changes for its AWS managed policies.	April 21, 2021

Amazon Timestream for LiveAnalytics identity-based policy examples

By default, IAM users and roles don't have permission to create or modify Timestream for LiveAnalytics resources. They also can't perform tasks using the AWS Management Console, CQLSH, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the *IAM User Guide*.

Topics

- Policy best practices
- Using the Timestream for LiveAnalytics console
- Allow users to view their own permissions
- Common operations in Timestream for LiveAnalytics
- Timestream for LiveAnalytics resource access based on tags
- Scheduled queries

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Timestream for LiveAnalytics resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.

- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Timestream for LiveAnalytics console

Timestream for LiveAnalytics does not require specific permissions to access the Amazon Timestream for LiveAnalytics console. You need at least read-only permissions to list and view details about the Timestream for LiveAnalytics resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (IAM users or roles) with that policy.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
```

```
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Common operations in Timestream for LiveAnalytics

Below are sample IAM policies that allow for common operations in the Timestream for LiveAnalytics service.

Topics

- Allowing all operations
- Allowing SELECT operations
- Allowing SELECT operations on multiple resources
- Allowing metadata operations
- Allowing INSERT operations
- Allowing CRUD operations
- Cancel queries and select data without specifying resources

- Create, describe, delete and describe a database
- Limit listed databases by tag{"Owner": "${username}"}
- List all tables in a database
- Create, describe, delete, update and select on a table
- Limit a query by table

Allowing all operations

The following is a sample policy that allows all operations in Timestream for LiveAnalytics.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:*"
            ],
            "Resource": "*"
        }
    ]
}
```

Allowing SELECT operations

The following sample policy allows SELECT-style queries on a specific resource.

> **Note**
>
> Replace <account_ID> with your Amazon account ID.

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:Select",
                "timestream:DescribeTable",
                "timestream:ListMeasures"
            ],
            "Resource": "arn:aws:timestream:us-east-1:111122223333:database/
sampleDB/table/DevOps"
        },
        {
            "Effect": "Allow",
            "Action": [
                "timestream:DescribeEndpoints",
                "timestream:SelectValues",
                "timestream:CancelQuery"
            ],
            "Resource": "*"
        }
    ]
}
```

Allowing SELECT operations on multiple resources

The following sample policy allows SELECT-style queries on multiple resources.

> **Note**
>
> Replace <account_ID> with your Amazon account ID.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "timestream:Select",
                "timestream:DescribeTable",
                "timestream:ListMeasures"
            ],
            "Resource": [
                "arn:aws:timestream:us-east-1:111122223333:database/sampleDB/
table/DevOps",
                "arn:aws:timestream:us-east-1:111122223333:database/sampleDB/
table/DevOps1",
                "arn:aws:timestream:us-east-1:111122223333:database/sampleDB/
table/DevOps2"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "timestream:DescribeEndpoints",
                "timestream:SelectValues",
                "timestream:CancelQuery"
            ],
            "Resource": "*"
        }
    ]
}
```

Allowing metadata operations

The following sample policy allows the user to perform metadata queries, but does not allow the user to perform operations that read or write actual data in Timestream for LiveAnalytics.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:DescribeEndpoints",
                "timestream:DescribeTable",
                "timestream:ListMeasures",
                "timestream:SelectValues",
```

```
                "timestream:ListTables",
                "timestream:ListDatabases",
                "timestream:CancelQuery"
            ],
            "Resource": "*"
        }
    ]
}
```

Allowing INSERT operations

The following sample policy allows a user to perform an `INSERT` operation on `database/sampleDB/table/DevOps` in account `<account_id>`.

> **Note**
>
> Replace `<account_ID>` with your Amazon account ID.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "timestream:WriteRecords"
            ],
            "Resource": [
                "arn:aws:timestream:us-east-1:111122223333:database/sampleDB/
table/DevOps"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "timestream:DescribeEndpoints"
            ],
            "Resource": "*",
            "Effect": "Allow"
        }
```

```
    ]
}
```

Allowing CRUD operations

The following sample policy allows a user to perform CRUD operations in Timestream for LiveAnalytics.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:DescribeEndpoints",
                "timestream:CreateTable",
                "timestream:DescribeTable",
                "timestream:CreateDatabase",
                "timestream:DescribeDatabase",
                "timestream:ListTables",
                "timestream:ListDatabases",
                "timestream:DeleteTable",
                "timestream:DeleteDatabase",
                "timestream:UpdateTable",
                "timestream:UpdateDatabase"
            ],
            "Resource": "*"
        }
    ]
}
```

Cancel queries and select data without specifying resources

The following sample policy allows a user to cancel queries and perform `Select` queries on data that does not require resource specification:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:SelectValues",
                "timestream:CancelQuery"
            ],
            "Resource": "*"
        }
    ]
}
```

Create, describe, delete and describe a database

The following sample policy allows a user to create, describe, delete and describe database `sampleDB`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:CreateDatabase",
                "timestream:DescribeDatabase",
                "timestream:DeleteDatabase",
                "timestream:UpdateDatabase"
            ],
            "Resource": "arn:aws:timestream:us-east-1:111122223333:database/
sampleDB"
        }
    ]
}
```

Limit listed databases by tag`{"Owner": "${username}"}`

The following sample policy allows a user to list all databases that that are tagged with key value pair `{"Owner": "${username}"}`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:ListDatabases"
            ],
            "Resource": "arn:aws:timestream:us-east-1:111122223333:database/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

List all tables in a database

The following sample policy to list all tables in database `sampleDB`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:ListTables"
            ],
            "Resource": "arn:aws:timestream:us-east-1:111122223333:database/
sampleDB/"
```

```
        }
    ]
}
```

Create, describe, delete, update and select on a table

The following sample policy allows a user to create tables, describe tables, delete tables, update tables, and perform `Select` queries on table `DevOps` in database `sampleDB`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:CreateTable",
                "timestream:DescribeTable",
                "timestream:DeleteTable",
                "timestream:UpdateTable",
                "timestream:Select"
            ],
            "Resource": "arn:aws:timestream:us-east-1:111122223333:database/
sampleDB/table/DevOps"
        }
    ]
}
```

Limit a query by table

The following sample policy allows a user to query all tables except `DevOps` in database `sampleDB`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "timestream:Select"
            ],
            "Resource": "arn:aws:timestream:us-east-1:111122223333:database/
sampleDB/table/*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "timestream:Select"
            ],
            "Resource": "arn:aws:timestream:us-east-1:111122223333:database/
sampleDB/table/DevOps"
        }
    ]
}
```

Timestream for LiveAnalytics resource access based on tags

You can use conditions in your identity-based policy to control access to Timestream for LiveAnalytics resources based on tags. This section provides some examples.

The following example shows how you can create a policy that grants permissions to a user to view a table if the table's `Owner` contains the value of that user's user name.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyAccessTaggedTables",
            "Effect": "Allow",
            "Action": "timestream:Select",
            "Resource": "arn:aws:timestream:us-east-2:111122223333:database/
mydatabase/table/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
```

```
            }
        }
    ]
}
```

You can attach this policy to the IAM users in your account. If a user named `richard-roe` attempts to view an Timestream for LiveAnalytics table, the table must be tagged `Owner=richard-roe` or `owner=richard-roe`. Otherwise, he is denied access. The condition tag key `Owner` matches both `Owner` and `owner` because condition key names are not case-sensitive. For more information, see IAM JSON Policy Elements: Condition in the *IAM User Guide*.

The following policy grants permissions to a user to create tables with tags if the tag passed in request has a key `Owner` and a value `username`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CreateTagTableUser",
            "Effect": "Allow",
            "Action": [
                "timestream:CreateTable",
                "timestream:TagResource"
            ],
            "Resource": "arn:aws:timestream:us-east-2:111122223333:database/
mydatabase/table/*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "aws:RequestTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

The policy below allows use of the `DescribeDatabase` API on any Database that has the `env` tag set to either `dev` or `test`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDescribe",
            "Effect": "Allow",
            "Action": [
                "timestream:DescribeEndpoints",
                "timestream:DescribeDatabase"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowTagAccessForDevResources",
            "Effect": "Allow",
            "Action": [
                "timestream:TagResource"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/env": [
                        "test",
                        "dev"
                    ]
                }
            }
        }
    ]
}
```

This policy uses a `Condition` key to allow a tag that has the key `env` and a value of `test`, `qa`, or `dev` to be added to a resource.

Scheduled queries

List, delete, update, execute ScheduledQuery

The following sample policy allows a user to list, delete, update and execute scheduled queries.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:DeleteScheduledQuery",
                "timestream:ExecuteScheduledQuery",
                "timestream:UpdateScheduledQuery",
                "timestream:ListScheduledQueries",
                "timestream:DescribeEndpoints"
            ],
            "Resource": "*"
        }
    ]
}
```

CreateScheduledQuery using a customer managed KMS key

The following sample policy allows a user to create a scheduled query that is encrypted using a customer managed KMS key; *<keyid for ScheduledQuery>*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/ScheduledQueryExecutionRole"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
```

```
                "timestream:CreateScheduledQuery",
                "timestream:DescribeEndpoints"
            ],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "arn:aws:kms:us-west-2:123456789012:key/<keyid for
 ScheduledQuery>",
            "Effect": "Allow"
        }
    ]
}
```

DescribeScheduledQuery using a customer managed KMS key

The following sample policy allows a user to describe a scheduled query that was created using a customer managed KMS key; *<keyid for ScheduledQuery>*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "timestream:DescribeScheduledQuery",
                "timestream:DescribeEndpoints"
            ],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-west-2:123456789012:key/<keyid for
 ScheduledQuery>",
```

```
            "Effect": "Allow"
        }
    ]
}
```

Execution role permissions (using a customer managed KMS key for scheduled query and SSE-KMS for error reports)

Attach the following sample policy to the IAM role specified in the `ScheduledQueryExecutionRoleArn` parameter, of the `CreateScheduledQuery` API that uses customer managed KMS key for the scheduled query encryption and `SSE-KMS` encryption for error reports.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "kms:GenerateDataKey"
            ],
            "Resource": "arn:aws:kms:us-west-2:123456789012:key/<keyid for
 ScheduledQuery>",
            "Effect": "Allow"
        },
        {
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": [
                "arn:aws:kms:us-west-2:123456789012:key/<keyid for database-1>",
                "arn:aws:kms:us-west-2:123456789012:key/<keyid for database-n>",
                "arn:aws:kms:us-west-2:123456789012:key/<keyid for
 ScheduledQuery>"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "sns:Publish"
```

```
            ],
            "Resource": [
                "arn:aws:sns:us-west-2:123456789012:scheduled-query-notification-
topic-*"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "timestream:Select",
                "timestream:SelectValues",
                "timestream:WriteRecords"
            ],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "s3:PutObject",
                "s3:GetBucketAcl"
            ],
            "Resource": [
                "arn:aws:s3:::scheduled-query-error-bucket",
                "arn:aws:s3:::scheduled-query-error-bucket/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

Execution role trust relationship

The following is the trust relationship for the IAM role specified in the `ScheduledQueryExecutionRoleArn` parameter of the `CreateScheduledQuery` API.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Principal": {
                "Service": [
                    "timestream.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Allow access to all scheduled queries created within an account

Attach the following sample policy to the IAM role specified in the `ScheduledQueryExecutionRoleArn` parameter, of the `CreateScheduledQuery` API, to allow access to all scheduled queries created within the an account *`Account_ID`*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "timestream.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "Account_ID"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:timestream:us-
west-2:111122223333:scheduled-query/*"
                }
            }
        }
    ]
}
```

Allow access to all scheduled queries with a specific name

Attach the following sample policy to the IAM role specified in the `ScheduledQueryExecutionRoleArn` parameter, of the `CreateScheduledQuery` API, to allow access to all scheduled queries with a name that starts with *`Scheduled_Query_Name`*, within account *`Account_ID`*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "timestream.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "Account_ID"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:timestream:us-
west-2:111122223333:scheduled-query/Scheduled_Query_Name*"
                }
            }
        }
    ]
}
```

Troubleshooting Amazon Timestream for LiveAnalytics identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Timestream for LiveAnalytics and IAM.

Topics

- I am not authorized to perform an action in Timestream for LiveAnalytics
- I am not authorized to perform iam:PassRole

- I want to allow people outside of my AWS account to access my Timestream for LiveAnalytics resources

I am not authorized to perform an action in Timestream for LiveAnalytics

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your sign-in credentials.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a *`table`* but does not have `timestream:`*`Select`* permissions for the table.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 timestream:Select on resource: mytable
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *`mytable`* resource using the `timestream:`*`Select`* action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Timestream for LiveAnalytics.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Timestream for LiveAnalytics. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Timestream for LiveAnalytics resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Timestream for LiveAnalytics supports these features, see How Amazon Timestream for LiveAnalytics works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Logging and monitoring in Timestream for LiveAnalytics

Monitoring is an important part of maintaining the reliability, availability, and performance of Timestream for LiveAnalytics and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. However, before you start monitoring Timestream for LiveAnalytics, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal Timestream for LiveAnalytics performance in your environment, by measuring performance at various times and under different load conditions. As you monitor Timestream for LiveAnalytics, store historical monitoring data so that you can compare it with current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish a baseline, you should, at a minimum, monitor the following items:

- System errors, so that you can determine whether any requests resulted in an error.

Topics

- Monitoring tools
- Logging Timestream for LiveAnalytics API calls with AWS CloudTrail

Monitoring tools

AWS provides various tools that you can use to monitor Timestream for LiveAnalytics. You can configure some of these tools to do the monitoring for you, while some of the tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Topics

- Automated monitoring tools
- Manual monitoring tools

Automated monitoring tools

You can use the following automated monitoring tools to watch Timestream for LiveAnalytics and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see Monitoring with Amazon CloudWatch.

Manual monitoring tools

Another important part of monitoring Timestream for LiveAnalytics involves manually monitoring those items that the CloudWatch alarms don't cover. The Timestream for LiveAnalytics, CloudWatch, Trusted Advisor, and other AWS Management Console dashboards provide an at-a-glance view of the state of your AWS environment.

- The CloudWatch home page shows the following:
  - Current alarms and status
  - Graphs of alarms and resources
  - Service health status

  In addition, you can use CloudWatch to do the following:
  - Create customized dashboards to monitor the services you care about
  - Graph metric data to troubleshoot issues and discover trends
  - Search and browse all your AWS resource metrics
  - Create and edit alarms to be notified of problems

Logging Timestream for LiveAnalytics API calls with AWS CloudTrail

Timestream for LiveAnalytics is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Timestream for LiveAnalytics. CloudTrail captures Data Definition Language (DDL) API calls for Timestream for LiveAnalytics as events. The calls that are captured include calls from the Timestream for LiveAnalytics console and code calls to the Timestream for LiveAnalytics API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for Timestream for LiveAnalytics. If you don't configure a trail, you can still view the most recent events on the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Timestream for LiveAnalytics, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Timestream for LiveAnalytics information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Timestream for LiveAnalytics, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

> **Warning**
>
> Currently, Timestream for LiveAnalytics generates CloudTrail events for all management and `Query` API operations, but does not generate events for `WriteRecords` and `DescribeEndpoints` APIs.

For an ongoing record of events in your AWS account, including events for Timestream for LiveAnalytics, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.

For more information, see the following topics in the *AWS CloudTrail User Guide*:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions
- Receiving CloudTrail Log Files from Multiple Accounts
- Logging data events

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials
- Whether the request was made with temporary security credentials for a role or federated user
- Whether the request was made by another AWS service

For more information, see the CloudTrail userIdentity Element.

For `Query` API events:

- Create a trail that receives all events or select events with Timestream for LiveAnalytics resource type `AWS::Timestream::Database` or `AWS::Timestream::Table`.
- `Query` API requests that do not access any database or table or that result in a validation exception due to a malformed query string are recorded in CloudTrail with a resource type `AWS::Timestream::Database` and an ARN value of:

  ```
  arn:aws:timestream:(region):(accountId):database/NO_RESOURCE_ACCESSED
  ```

  These events are delivered only to trails that receive events with resource type `AWS::Timestream::Database`.

Resilience in Amazon Timestream Live Analytics

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

For information about data protection functionality for Timestream available through AWS Backup, see Working with AWS Backup.

Infrastructure security in Amazon Timestream Live Analytics

As a managed service, Amazon Timestream Live Analytics is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Timestream Live Analytics through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-

Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Timestream Live Analytics is architected so that your traffic is isolated to the specific AWS Region that your Timestream Live Analytics instance resides in.

Configuration and vulnerability analysis in Timestream

Configuration and IT controls are a shared responsibility between AWS and you, our customer. For more information, see the AWS shared responsibility model. In addition to the shared responsibility model, Timestream for LiveAnalytics users should be aware of the following:

- It is the customer responsibility to patch their client applications with the relevant client side dependencies.
- Customers should consider penetration testing if appropriate (see https://aws.amazon.com/security/penetration-testing/.)

Incident response in Timestream for LiveAnalytics

Amazon Timestream for LiveAnalytics service incidents are reported in the Personal Health Dashboard. You can learn more about the dashboard and AWS Health here.

Timestream for LiveAnalytics supports reporting using AWS CloudTrail. For more information, see Logging Timestream for LiveAnalytics API calls with AWS CloudTrail.

VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon Timestream for LiveAnalytics by creating an *interface VPC endpoint*. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to privately access Timestream for LiveAnalytics APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with Timestream for LiveAnalytics APIs. Traffic between your VPC and Timestream for LiveAnalytics does not leave the Amazon network.

Each interface endpoint is represented by one or more Elastic Network Interfaces in your subnets. For more information on Interface VPC endpoints, see Interface VPC endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*.

To get started with Timestream for LiveAnalytics and VPC endpoints, we've provided information on specific considerations for Timestream for LiveAnalytics with VPC endpoints, creating an interface VPC endpoint for Timestream for LiveAnalytics, creating a VPC endpoint policy for Timestream for LiveAnalytics, and using the Timestream client (for either the Write or Query SDK) with VPC endpoints..

Topics

- How VPC endpoints work with Timestream
- Creating an interface VPC endpoint for Timestream for LiveAnalytics
- Creating a VPC endpoint policy for Timestream for LiveAnalytics

How VPC endpoints work with Timestream

When you create a VPC endpoint to access either the Timestream Write or Timestream Query SDK, all requests are routed to endpoints within the Amazon network and do not access the public internet. More specifically, your requests are routed to the write and query endpoints of the cell that your account has been mapped to for a given region. To learn more about Timestream's cellular architecture and cell-specific endpoints, you can refer to Cellular architecture. For example, suppose that your account has been mapped to `cell1` in `us-west-2`, and you've set up VPC interface endpoints for writes (`ingest-cell1.timestream.us-west-2.amazonaws.com`) and queries (`query-cell1.timestream.us-west-2.amazonaws.com`). In this case, any write requests sent using these endpoints will stay entirely within the Amazon network and will not access the public internet.

Considerations for Timestream VPC endpoints

Consider the following when creating a VPC endpoint for Timestream:

- Before you set up an interface VPC endpoint for Timestream for LiveAnalytics, ensure that you review Interface endpoint properties and limitations in the *Amazon VPC User Guide*.
- Timestream for LiveAnalytics supports making calls to all of its API actions from your VPC.

- VPC endpoint policies are supported for Timestream for LiveAnalytics. By default, full access to Timestream for LiveAnalytics is allowed through the endpoint. For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.
- Because of Timestream's architecture, access to both Write and Query actions requires the creation of two VPC interface endpoints, one for each SDK. Additionally, you must specify a cell endpoint (you will only be able to create an endpoint for the Timestream cell that you are mapped to). Detailed information can be found in the create an interface VPC endpoint for Timestream for LiveAnalytics section of this guide.

Now that you understand how Timestream for LiveAnalytics works with VPC endpoints, create an interface VPC endpoint for Timestream for LiveAnalytics.

Creating an interface VPC endpoint for Timestream for LiveAnalytics

You can create an interface VPC endpoint for the Timestream for LiveAnalytics service using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). To create a VPC endpoint for Timestream, complete the Timestream-specific steps described below.

> **Note**
>
> Before completing the steps below, ensure that you understand specific considerations for Timestream VPC endpoints.

Constructing a VPC endpoint service name using your Timestream cell

Because of Timestream's unique architecture, separate VPC interface endpoints must be created for each SDK (Write and Query). Additionally, you must specify a Timestream cell endpoint (you will only be able to create an endpoint for the Timestream cell that you are mapped to). To use Interface VPC Endpoints to directly connect to Timestream from within your VPC, complete the steps below:

1. First, find an available Timestream cell endpoint. To find an available cell endpoint, use the `DescribeEndpoints` action (available through both the Write and Query APIs) to list the cell endpoints available in your Timestream account. See the example for further details.
2. Once you've selected a cell endpoint to use, create a VPC interface endpoint string for either the Timestream Write or Query API:
   - *For the Write API:*

```
com.amazonaws.<region>.timestream.ingest-<cell>
```

- *For the Query API:*

```
com.amazonaws.<region>.timestream.query-<cell>
```

where *<region>* is a valid AWS region code and *<cell>* is one of the cell endpoint addresses (such as `cell1` or `cell2`) returned in the Endpoints object by the DescribeEndpoints action. See the example for further details.

3. Now that you have constructed a VPC endpoint service name, create an interface endpoint. When asked to provide a VPC endpoint service name, use the VPC endpoint service name that you constructed in Step 2.

Example: Constructing your VPC endpoint service name

In the following example, the `DescribeEndpoints` action is executed in the AWS CLI using the Write API in the `us-west-2` region:

```
aws timestream-write describe-endpoints --region us-west-2
```

This command will return the following output:

```
{
    "Endpoints": [
        {
            "Address": "ingest-cell1.timestream.us-west-2.amazonaws.com",
            "CachePeriodInMinutes": 1440
        }
    ]
}
```

In this case, *cell1* is the *<cell>* , and *us-west-2* is the *<region>*. So, the resulting VPC endpoint service name will look like:

```
com.amazonaws.us-west-2.timestream.ingest-cell1
```

Now that you've created an interface VPC endpoint for Timestream for LiveAnalytics, create a VPC endpoint policy for Timestream for LiveAnalytics.

Creating a VPC endpoint policy for Timestream for LiveAnalytics

You can attach an endpoint policy to your VPC endpoint that controls access to Timestream for LiveAnalytics. The policy specifies the following information:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Example: VPC endpoint policy for Timestream for LiveAnalytics actions

The following is an example of an endpoint policy for Timestream for LiveAnalytics. When attached to an endpoint, this policy grants access to the listed Timestream for LiveAnalytics actions (in this case, `ListDatabases`) for all principals on all resources.

```
{
   "Statement":[
      {
         "Principal":"*",
         "Effect":"Allow",
         "Action":[
            "timestream:ListDatabases"
         ],
         "Resource":"*"
      }
   ]
}
```

Security best practices for Amazon Timestream for LiveAnalytics

Amazon Timestream for LiveAnalytics provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Topics

- Timestream for LiveAnalytics preventative security best practices

Timestream for LiveAnalytics preventative security best practices

The following best practices can help you anticipate and prevent security incidents in Timestream for LiveAnalytics.

Encryption at rest

Timestream for LiveAnalytics encrypts at rest all user data stored in tables using encryption keys stored in AWS Key Management Service (AWS KMS). This provides an additional layer of data protection by securing your data from unauthorized access to the underlying storage.

Timestream for LiveAnalytics uses a single service default key (AWS owned CMK) for encrypting all of your tables. If this key doesn't exist, it is created for you. Service default keys can't be disabled. For more information, see Timestream for LiveAnalytics Encryption at Rest.

Use IAM roles to authenticate access to Timestream for LiveAnalytics

For users, applications, and other AWS services to access Timestream for LiveAnalytics, they must include valid AWS credentials in their AWS API requests. You should not store AWS credentials directly in the application or EC2 instance. These are long-term credentials that are not automatically rotated, and therefore could have significant business impact if they are compromised. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources.

For more information, see IAM Roles.

Use IAM policies for Timestream for LiveAnalytics base authorization

When granting permissions, you decide who is getting them, which Timestream for LiveAnalytics APIs they are getting permissions for, and the specific actions you want to allow on those resources. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

Attach permissions policies to IAM identities (that is, users, groups, and roles) and thereby grant permissions to perform operations on Timestream for LiveAnalytics resources.

You can do this by using the following:

- AWS managed (predefined) policies

- Customer managed policies
- Tag-based authorization

Consider client-side encryption

If you store sensitive or confidential data in Timestream for LiveAnalytics, you might want to encrypt that data as close as possible to its origin so that your data is protected throughout its lifecycle. Encrypting your sensitive data in transit and at rest helps ensure that your plaintext data isn't available to any third party.

Working with other services

Amazon Timestream for LiveAnalytics integrates with a variety of AWS services and popular third-party tools. Currently, Timestream for LiveAnalytics supports integrations with the following:

Topics

- Amazon DynamoDB
- AWS Lambda
- AWS IoT Core
- Amazon Managed Service for Apache Flink
- Amazon Kinesis
- Amazon MQ
- Amazon MSK
- Amazon Quick
- Amazon SageMaker AI
- Amazon SQS
- Using DBeaver to work with Amazon Timestream
- Grafana
- Using SquaredUp to work with Amazon Timestream
- Open source Telegraf
- JDBC
- ODBC

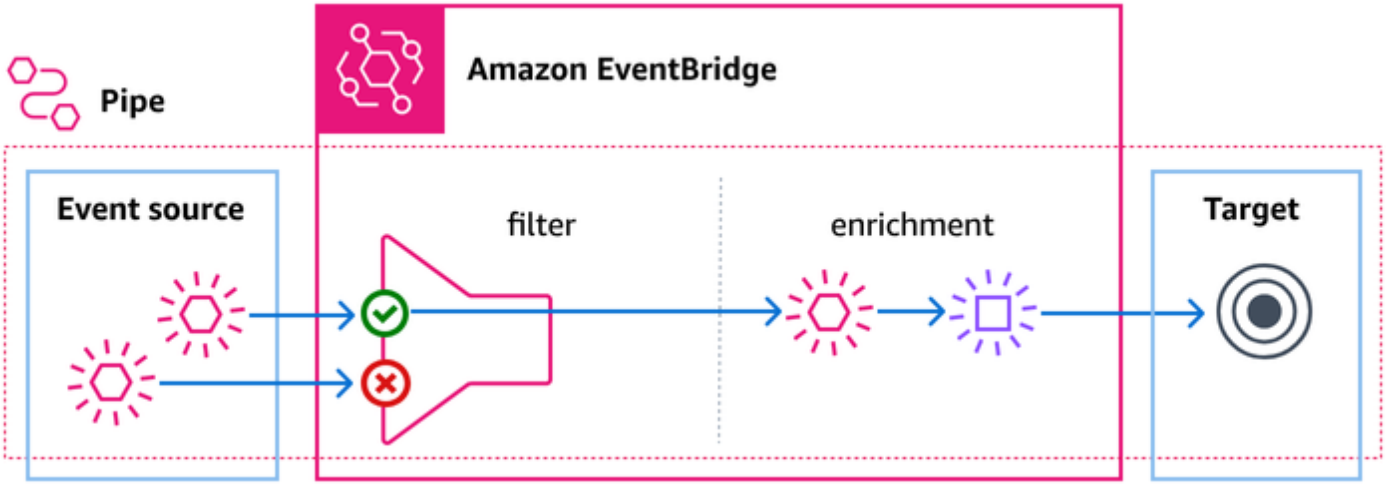
- VPC endpoints (AWS PrivateLink)

Amazon DynamoDB

Using EventBridge Pipes to send DynamoDB data to Timestream

You can use EventBridge Pipes to send data from a DynamoDB stream to a Amazon Timestream for LiveAnalytics table.

Pipes are intended for point-to-point integrations between supported sources and targets, with support for advanced transformations and enrichment. Pipes reduce the need for specialized knowledge and integration code when developing event-driven architectures. To set up a pipe, you choose the source, add optional filtering, define optional enrichment, and choose the target for the event data.



For more information on EventBridge Pipes, see EventBridge Pipes in the *EventBridge User Guide*. For information on configuring a pipe to deliver events to a Amazon Timestream for LiveAnalytics table, see EventBridge Pipes target specifics.

AWS Lambda

You can create Lambda functions that interact with Timestream for LiveAnalytics. For example, you can create a Lambda function that runs at regular intervals to execute a query on Timestream and send an SNS notification based on the query results satisfying one or more criteria. To learn more about Lambda, see the AWS Lambda documentation.

Topics

- Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with Python
- Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with JavaScript
- Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with Go
- Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with C#

Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with Python

To build AWS Lambda functions using Amazon Timestream for LiveAnalytics with Python, follow the steps below.

1. Create an IAM role for Lambda to assume that will grant the required permissions to access the Timestream Service, as outlined in Provide Timestream for LiveAnalytics access.
2. Edit the trust relationship of the IAM role to add Lambda service. You can use the commands below to update an existing role so that AWS Lambda can assume it:

   a. Create the trust policy document:

   ```
   cat > Lambda-Role-Trust-Policy.json << EOF
   {
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
           "Service": [
             "lambda.amazonaws.com"
           ]
         },
         "Action": "sts:AssumeRole"
       }
     ]
   }
   EOF
   ```

   b. Update the role from previous step with the trust document

   ```
   aws iam update-assume-role-policy --role-name <name_of_the_role_from_step_1> --
   policy-document file://Lambda-Role-Trust-Policy.json
   ```

Related references are at TimestreamWrite and TimestreamQuery.

Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with JavaScript

To build AWS Lambda functions using Amazon Timestream for LiveAnalytics with JavaScript, follow the instructions outlined here.

Related references are at Timestream Write Client - AWS SDK for JavaScript v3 and Timestream Query Client - AWS SDK for JavaScript v3.

Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with Go

To build AWS Lambda functions using Amazon Timestream for LiveAnalytics with Go, follow the instructions outlined here.

Related references are at timestreamwrite and timestreamquery.

Build AWS Lambda functions using Amazon Timestream for LiveAnalytics with C#

To build AWS Lambda functions using Amazon Timestream for LiveAnalytics with C#, follow the instructions outlined here.

Related references are at Amazon.TimestreamWrite and Amazon.TimestreamQuery.

AWS IoT Core

You can collect data from IoT devices using AWS IoT Core and route the data to Amazon Timestream through IoT Core rule actions. AWS IoT rule actions specify what to do when a rule is triggered. You can define actions to send data to an Amazon Timestream table, an Amazon DynamoDB database, and invoke an AWS Lambda function.

The Timestream action in IoT Rules is used to insert data from incoming messages directly into Timestream. The action parses the results of the IoT Core SQL statement and stores data in Timestream. The names of the fields from returned SQL result set are used as the measure::name and the value of the field is the measure::value.

For example, consider the SQL statement and the sample message payload:

```
SELECT temperature, humidity from 'iot/topic'
```

```
{
  "dataFormat": 5,
  "rssi": -88,
  "temperature": 24.04,
  "humidity": 43.605,
  "pressure": 101082,
  "accelerationX": 40,
  "accelerationY": -20,
  "accelerationZ": 1016,
  "battery": 3007,
  "txPower": 4,
  "movementCounter": 219,
  "device_id": 46216,
  "device_firmware_sku": 46216
}
```

If an IoT Core rule action for Timestream is created with the SQL statement above, two records will be added to Timestream with measure names temperature and humidity and measure values of 24.04 and 43.605, respectively.

You can modify the measure name of a record being added to Timestream by using the AS operator in the SELECT statement. The SQL statement below will create a record with the message name temp instead of temperature.

The data type of the measure are inferred from the data type of the value of the message payload. JSON data types such as integer, double, boolean, and string are mapped to Timestream data types of BIGINT, DOUBLE, BOOLEAN, and VARCHAR respectively. Data can also be forced to specific data types using the cast() function. You can specify the timestamp of the measure. If the timestamp is left blank, the time that the entry was processed is used.

You can refer to the Timestream rules action documentation for additional details

To create an IoT Core rule action to store data in Timestream, follow the steps below:

Topics

- Prerequisites
- Using the console
- Using the CLI
- Sample application
- Video tutorial

Prerequisites

1. Create a database in Amazon Timestream using the instructions described in Create a database.
2. Create a table in Amazon Timestream using the instructions described in Create a table.

Using the console

1. Use the AWS Management Console for AWS IoT Core to create a rule by clicking on **Manage** > **Messsage routing** > **Rules** followed by **Create rule**.
2. Set the rule name to a name of your choice and the SQL to the text shown below

```
SELECT temperature as temp, humidity from 'iot/topic'
```

3. Select Timestream from the Action list
4. Specify the Timestream database, table, and dimension names along with the role to write data into Timestream. If the role does not exist, you can create one by clicking on Create Roles
5. To test the rule, follow the instructions shown here.

Using the CLI

If you haven't installed the AWS Command Line Interface (AWS CLI), do so from here.

1. Save the following rule payload in a JSON file called timestream_rule.json. Replace *arn:aws:iam::123456789012:role/TimestreamRole* with your role arn which grants AWS IoT access to store data in Amazon Timestream

```
{
    "actions": [
            {
                "timestream": {
                    "roleArn": "arn:aws:iam::123456789012:role/TimestreamRole",
                    "tableName": "devices_metrics",
                    "dimensions": [
                        {
                            "name": "device_id",
                            "value": "${clientId()}"
                        },
```

```
                        {
                            "name": "device_firmware_sku",
                            "value": "My Static Metadata"
                        }
                    ],
                    "databaseName": "record_devices"
                }
            }
    ],
    "sql": "select * from 'iot/topic'",
    "awsIotSqlVersion": "2016-03-23",
    "ruleDisabled": false
}
```

2. Create a topic rule using the following command

```
aws iot create-topic-rule --rule-name timestream_test --topic-rule-payload file://
<path/to/timestream_rule.json> --region us-east-1
```

3. Retrieve details of topic rule using the following command

```
aws iot get-topic-rule --rule-name timestream_test
```

4. Save the following message payload in a file called timestream_msg.json

```
{
  "dataFormat": 5,
  "rssi": -88,
  "temperature": 24.04,
  "humidity": 43.605,
  "pressure": 101082,
  "accelerationX": 40,
  "accelerationY": -20,
  "accelerationZ": 1016,
  "battery": 3007,
  "txPower": 4,
  "movementCounter": 219,
  "device_id": 46216,
  "device_firmware_sku": 46216
}
```

5. Test the rule using the following command

```
aws iot-data publish --topic 'iot/topic' --payload file://<path/to/
timestream_msg.json>
```

Sample application

To help you get started with using Timestream with AWS IoT Core, we've created a fully functional sample application that creates the necessary artifacts in AWS IoT Core and Timestream for creating a topic rule and a sample application for publishing a data to the topic.

1. Clone the GitHub repository for the sample application for AWS IoT Core integration following the instructions from GitHub
2. Follow the instructions in the README to use an AWS CloudFormation template to create the necessary artifacts in Amazon Timestream and AWS IoT Core and to publish sample messages to the topic.

Video tutorial

This video explains how IoT Core works with Timestream.

Amazon Managed Service for Apache Flink

You can use Apache Flink to transfer your time series data from Amazon Managed Service for Apache Flink, Amazon MSK, Apache Kafka, and other streaming technologies directly into Amazon Timestream for LiveAnalytics. We've created an Apache Flink sample data connector for Timestream. We've also created a sample application for sending data to Amazon Kinesis so that the data can flow from Kinesis to Managed Service for Apache Flink, and finally on to Amazon Timestream. All of these artifacts are available to you in GitHub. This video tutorial describes the setup.

> **Note**
>
> Java 11 is the recommended version for using the Managed Service for Apache Flink Application. If you have multiple Java versions, ensure that you export Java 11 to your JAVA_HOME environment variable.

Topics

- Sample application
- Video tutorial

Sample application

To get started, follow the procedure below:

1. Create a database in Timestream with the name `kdaflink` following the instructions described in Create a database.
2. Create a table in Timestream with the name `kinesisdata1` following the instructions described in Create a table.
3. Create an Amazon Kinesis Data Stream with the name `TimestreamTestStream` following the instructions described in Creating a Stream.
4. Clone the GitHub repository for the Apache Flink data connector for Timestream following the instructions from GitHub.
5. To compile, run and use the sample application, follow the instructions in the Apache Flink sample data connector README.
6. Compile the Managed Service for Apache Flink application following the instructions for Compiling the Application Code.
7. Upload the Managed Service for Apache Flink application binary following the instructions to Upload the Apache Flink Streaming Code.

   a. After clicking on Create Application, click on the link of the IAM Role for the application.

   b. Attach the IAM policies for **AmazonKinesisReadOnlyAccess** and **AmazonTimestreamFullAccess**.

   > **Note**
   >
   > The above IAM policies are not restricted to specific resources and are unsuitable for production use. For a production system, consider using policies that restrict access to specific resources.

8. Clone the GitHub repository for the sample application writing data to Kinesis following the instructions from GitHub.
9. Follow the instructions in the README to run the sample application for writing data to Kinesis.

10. Run one or more queries in Timestream to ensure that data is being sent from Kinesis to Managed Service for Apache Flink to Timestream following the instructions to Create a table.

Video tutorial

This video explains how to use Timestream with Managed Service for Apache Flink.

Amazon Kinesis

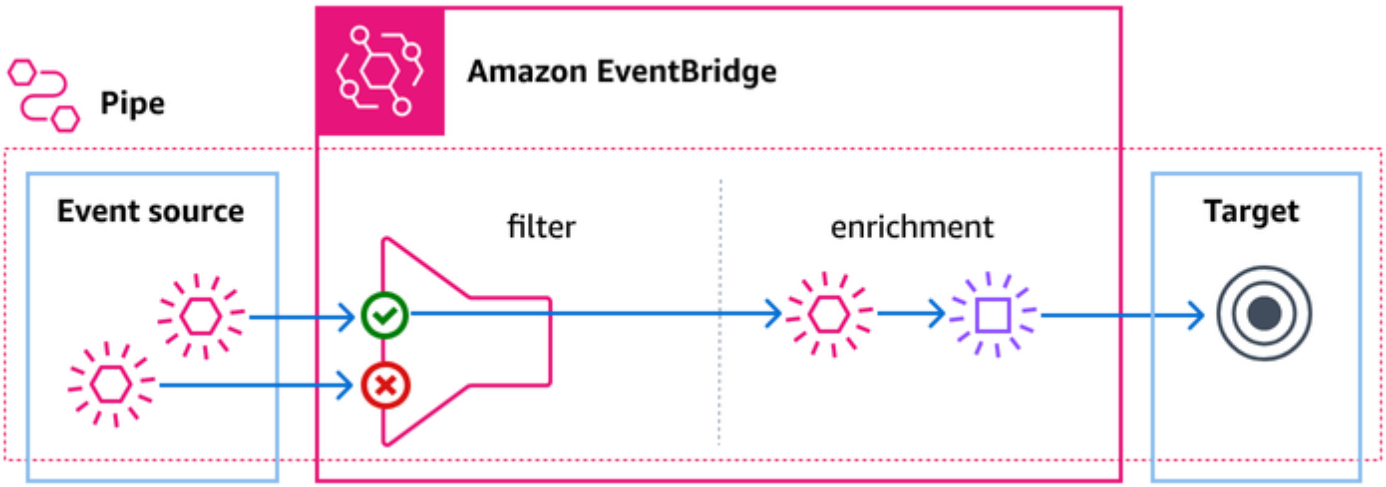
Using Amazon Managed Service for Apache Flink

You can send data from Kinesis Data Streams to Timestream for LiveAnalytics using the sample Timestream data connector for Managed Service for Apache Flink. Refer to Amazon Managed Service for Apache Flink for Apache Flink for more information.

Using EventBridge Pipes to send Kinesis data to Timestream

You can use EventBridge Pipes to send data from a Kinesis stream to a Amazon Timestream for LiveAnalytics table.

Pipes are intended for point-to-point integrations between supported sources and targets, with support for advanced transformations and enrichment. Pipes reduce the need for specialized knowledge and integration code when developing event-driven architectures. To set up a pipe, you choose the source, add optional filtering, define optional enrichment, and choose the target for the event data.



This integration enables you to leverage the power of Timestream's time-series data analysis capabilities, while simplifying your data ingestion pipeline.

Using EventBridge Pipes with Timestream offers the following benefits:

- Real-time Data Ingestion: Stream data from Kinesis directly to Timestream for LiveAnalytics, enabling real-time analytics and monitoring.
- Seamless Integration: Utilize EventBridge Pipes to manage the flow of data without the need for complex custom integrations.
- Enhanced Filtering and Transformation: Filter or transform Kinesis records before they are stored in Timestream to meet your specific data processing requirements.
- Scalability: Handle high-throughput data streams and ensure efficient data processing with built-in parallelism and batching capabilities.

Configuration

To set up an EventBridge Pipe to stream data from Kinesis to Timestream, follow these steps:

1. Create a Kinesis stream

   Ensure you have an active Kinesis data stream from which you want to ingest data.
2. Create a Timestream database and table

   Set up your Timestream database and table where the data will be stored.
3. Configure the EventBridge Pipe:
   - Source: Select your Kinesis stream as the source.
   - Target: Choose Timestream as the target.
   - Batching Settings: Define batching window and batch size to optimize data processing and reduce latency.

> **Important**
>
> When setting up a pipe, we recommend testing the correctness of all configurations by ingesting a few records. Please note that successful creation of a pipe does not guarantee that the pipeline is correct and data will flow without errors. There may be runtime errors,

such as incorrect table, incorrect dynamic path parameter, or invalid Timestream record after applying mapping, that will be discovered when actual data flows through the pipe.

The following configurations determine the rate at which data is ingested:

- BatchSize: The maximum size of the batch that will be sent to Timestream for LiveAnalytics. Range: 0 - 100. Recommendation is to keep this value as 100 to get maximum throughput.
- MaximumBatchingWindowInSeconds: The maximum time to wait to fill the batchSize before the batch is sent to Timestream for LiveAnalytics target. Depending on the rate of incoming events, this configuration will decide the delay of ingestion, recommendation is to keep this value < 10s to keep sending the data to Timestream in near real-time.
- ParallelizationFactor: The number of batches to process concurrently from each shard. Recommendation is to use the maximum value of 10 to get maximum throughput and near real-time ingestion.

  If your stream is read by multiple targets, use enhanced fan-out to provide a dedicated consumer to your pipe to achieve high throughput. For more information, see Developing enhanced fan-out consumers with the Kinesis Data Streams API in the *Kinesis Data Streams User Guide*.

> **Note**
>
> The maximum throughput that can be achieved is bounded by concurrent pipe executions per account.

The following configuration ensures prevention of data loss:

- DeadLetterConfig: Recommendation is to always configure DeadLetterConfig to avoid any data loss for cases when events could not be ingested to Timestream for LiveAnalytics due to user errors.

Optimize your pipe's performance with the following configuration settings, which helps prevent records from causing slowdowns or blockages.

- MaximumRecordAgeInSeconds: Records older than this will not be processed and will directly get moved to DLQ. We recommend setting this value to be no higher than the configured Memory store retention period of the target Timestream table.
- MaximumRetryAttempts: The number of retry attempts for a record before the record is sent to DeadLetterQueue. Recommendation is to configure this at 10. This should be able to help address any transient issues and for persistent issues, the record will be moved to DeadLetterQueue and unblock the rest of the stream.
- OnPartialBatchItemFailure: For sources that support partial batch processing, we recommend you to enable this and configure it as AUTOMATIC_BISECT for additional retry of failed records before dropping/sending to DLQ.

Configuration example

Here is an example of how to configure an EventBridge Pipe to stream data from a Kinesis stream to a Timestream table:

Example IAM policy updates for Timestream

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:WriteRecords"
            ],
            "Resource": [
                "arn:aws:timestream:us-east-1:123456789012:database/my-database/
table/my-table"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "timestream:DescribeEndpoints"
            ],
            "Resource": "*"
        }
```

```
    ]
}
```

Example Kinesis stream configuration

```
{
  "Source": "arn:aws:kinesis:us-east-1:123456789012:stream/my-kinesis-stream",
  "SourceParameters": {
    "KinesisStreamParameters": {
        "BatchSize": 100,
        "DeadLetterConfig": {
            "Arn": "arn:aws:sqs:us-east-1:123456789012:my-sqs-queue"
        },
       "MaximumBatchingWindowInSeconds": 5,
        "MaximumRecordAgeInSeconds": 1800,
        "MaximumRetryAttempts": 10,
        "StartingPosition": "LATEST",
       "OnPartialBatchItemFailure": "AUTOMATIC_BISECT"
    }
  }
}
```

Example Timestream target configuration

```
{
    "Target": "arn:aws:timestream:us-east-1:123456789012:database/my-database/table/my-
table",
    "TargetParameters": {
        "TimestreamParameters": {
            "DimensionMappings": [
                {
                    "DimensionName": "sensor_id",
                    "DimensionValue": "$.data.device_id",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "DimensionName": "sensor_type",
                    "DimensionValue": "$.data.sensor_type",
                    "DimensionValueType": "VARCHAR"
                },
                {
                    "DimensionName": "sensor_location",
```

```
                    "DimensionValue": "$.data.sensor_loc",
                    "DimensionValueType": "VARCHAR"
                }
            ],
            "MultiMeasureMappings": [
                {
                    "MultiMeasureName": "readings",
                    "MultiMeasureAttributeMappings": [
                        {
                            "MultiMeasureAttributeName": "temperature",
                            "MeasureValue": "$.data.temperature",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
                            "MultiMeasureAttributeName": "humidity",
                            "MeasureValue": "$.data.humidity",
                            "MeasureValueType": "DOUBLE"
                        },
                        {
                            "MultiMeasureAttributeName": "pressure",
                            "MeasureValue": "$.data.pressure",
                            "MeasureValueType": "DOUBLE"
                        }
                    ]
                }
            ],
            "SingleMeasureMappings": [],
            "TimeFieldType": "TIMESTAMP_FORMAT",
            "TimestampFormat": "yyyy-MM-dd HH:mm:ss.SSS",
            "TimeValue": "$.data.time",
            "VersionValue": "$.approximateArrivalTimestamp"
        }
    }
}
```

Event transformation

EventBridge Pipes allow you to transform data before it reaches Timestream. You can define transformation rules to modify the incoming Kinesis records, such as changing field names.

Suppose your Kinesis stream contains temperature and humidity data. You can use an EventBridge transformation to rename these fields before inserting them into Timestream.

Best practices

Batching and Buffering

- Configure the batching window and size to balance between write latency and processing efficiency.
- Use a batching window to accumulate enough data before processing, reducing the overhead of frequent small batches.

Parallel Processing

Utilize the **ParallelizationFactor** setting to increase concurrency, especially for high-throughput streams. This ensures that multiple batches from each shard can be processed simultaneously.

Data Transformation

Leverage the transformation capabilities of EventBridge Pipes to filter and enhance records before storing them in Timestream. This can help in aligning the data with your analytical requirements.

Security

- Ensure that the IAM roles used for EventBridge Pipes have the necessary permissions to read from Kinesis and write to Timestream.
- Use encryption and access control measures to secure data in transit and at rest.

Debugging failures

- **Automatic Disabling of Pipes**

  Pipes will be automatically disabled in about 2 hours if the target does not exist or has permission issues
- **Throttles**

  Pipes have the capability to automatically back off and retry until the throttles have reduced.
- **Enabling Logs**

  We recommend you enable Logs at ERROR level and include execution data to get more insights into failed. Upon any failure, these logs will contain request/response sent/received from

Timestream. This helps you understand the error associated and if needed reprocess the records after fixing it.

Monitoring

We recommend you to set up alarms on the following to detect any issues with data flow:

- Maximum Age of the Record in Source
  - `GetRecords.IteratorAgeMilliseconds`
- Failure metrics in Pipes
  - `ExecutionFailed`
  - `TargetStageFailed`
- Timestream Write API errors
  - `UserErrors`

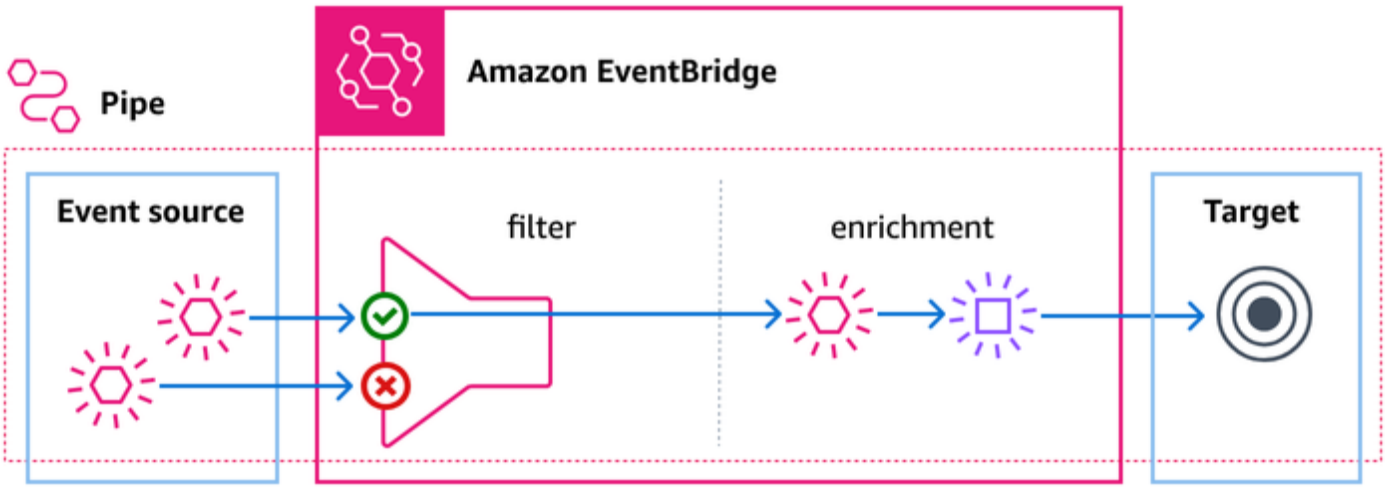
For additional monitoring metrics, see Monitoring EventBridge in the *EventBridge User Guide*.

Amazon MQ

Using EventBridge Pipes to send Amazon MQ data to Timestream

You can use EventBridge Pipes to send data from a Amazon MQ broker to a Amazon Timestream for LiveAnalytics table.

Pipes are intended for point-to-point integrations between supported sources and targets, with support for advanced transformations and enrichment. Pipes reduce the need for specialized knowledge and integration code when developing event-driven architectures. To set up a pipe, you choose the source, add optional filtering, define optional enrichment, and choose the target for the event data.



For more information on EventBridge Pipes, see EventBridge Pipes in the *EventBridge User Guide*. For information on configuring a pipe to deliver events to a Amazon Timestream for LiveAnalytics table, see EventBridge Pipes target specifics.

Amazon MSK

Using Managed Service for Apache Flink to send Amazon MSK data to Timestream for LiveAnalytics

You can send data from Amazon MSK to Timestream by building a data connector similar to the sample Timestream data connector for Managed Service for Apache Flink. Refer to Amazon Managed Service for Apache Flink for more information.

Using Kafka Connect to send Amazon MSK data to Timestream for LiveAnalytics

You can use Kafka Connect to ingest your time series data from Amazon MSK directly into Timestream for LiveAnalytics.

We've created a sample Kafka Sink Connector for Timestream. We've also created a sample Apache jMeter test plan for publishing data to a Kafka topic, so that the data can flow from the topic through the Timestream Kafka Sink Connector, to an Timestream for LiveAnalytics table. All of these artifacts are available on GitHub.

> **Note**
>
> Java 11 is the recommended version for using the Timestream Kafka Sink Connector. If you have multiple Java versions, ensure that you export Java 11 to your JAVA_HOME environment variable.

Creating a sample application

To get started, follow the procedure below.

1. In Timestream for LiveAnalytics, create a database with the name `kafkastream`.

   See the procedure ??? for detailed instructions.
2. In Timestream for LiveAnalytics, create a table with the name `purchase_history`.

   See the procedure ??? for detailed instructions.
3. Follow the instructions shared in the to create the following: , and .

   - An Amazon MSK cluster
   - An Amazon EC2 instance that is configured as a Kafka producer client machine
   - A Kafka topic

   See the prerequisites of the kafka_ingestor project for detailed instructions.
4. Clone the Timestream Kafka Sink Connector repository.

   See Cloning a repository on GitHub for detailed instructions.
5. Compile the plugin code.

   See Connector - Build from source on GitHub for detailed instructions.
6. Upload the following files to an S3 bucket: following the instructions described in .

   - The jar file (kafka-connector-timestream->VERSION<-jar-with-dependencies.jar) from the `/target` directory
   - The sample json schema file, `purchase_history.json`.

   See Uploading objects in the *Amazon S3 User Guide* for detailed instructions.

7. Create two VPC endpoints. These endpoints would be used by the MSK Connector to access the resources using AWS PrivateLink.

   - One to access the Amazon S3 bucket
   - One to access the Timestream for LiveAnalytics table.

   See VPC Endpoints for detailed instructions.

8. Create a custom plugin with the uploaded jar file.

   See Plugins in the *Amazon MSK Developer Guide* for detailed instructions.

9. Create a custom worker configuration with the JSON content described in Worker Configuration parameters. following the instructions described in

   See Creating a custom worker configuration in the *Amazon MSK Developer Guide* for detailed instructions.

10. Create a service execution IAM role.

    See IAM Service Role for detailed instructions.

11. Create an Amazon MSK connector with the custom plugin, custom worker configuration, and service execution IAM role created in the previous steps and with the Sample Connector Configuration.

    See Creating a connector in the *Amazon MSK Developer Guide* for detailed instructions.

    Make sure to update the values of the below configuration parameters with respective values. See Connector Configuration parameters for details.

    - `aws.region`
    - `timestream.schema.s3.bucket.name`
    - `timestream.ingestion.endpoint`

    The connector creation takes 5–10 minutes to complete. The pipeline is ready when its status changes to `Running`.

12. Publish a continuous stream of messages for writing data to the Kafka topic created.

    See How to use it for detailed instructions.

13. Run one or more queries to ensure that the data is being sent from Amazon MSK to MSK Connect to the Timestream for LiveAnalytics table.

    See the procedure ??? for detailed instructions.

Additional resources

The blog, Real-time serverless data ingestion from your Kafka clusters into Timestream for LiveAnalytics using Kafka Connect explains setting up an end-to-end pipeline using the Timestream for LiveAnalytics Kafka Sink Connector, starting from a Kafka producer client machine that uses the Apache jMeter test plan to publish thousands of sample messages to a Kafka topic to verifying the ingested records in an Timestream for LiveAnalytics table.

Amazon Quick

You can use Amazon Quick to analyze and publish data dashboards that contain your Amazon Timestream data. This section describes how you can create a new QuickSight data source connection, modify permissions, create new datasets, and perform an analysis. This video tutorial describes how to work with Timestream and Quick.

> **Note**
>
> All datasets in Quick are read-only. You can't make any changes to your actual data in Timestream by using Quick to remove the data source, dataset, or fields.

Topics

- Accessing Amazon Timestream from QuickSight
- Create a new QuickSight data source connection for Timestream
- Edit permissions for the QuickSight data source connection for Timestream
- Create a new QuickSight dataset for Timestream
- Create a new analysis for Timestream
- Video tutorial

Accessing Amazon Timestream from QuickSight

Before you can proceed, Amazon QuickSight needs to be authorized to connect to Amazon Timestream. If connections are not enabled, you will receive an error when you try to connect. A QuickSight administrator can authorize connections to AWS resources. To authorize a connection from QuickSight to Timestream, follow the procedure at Using Other AWS Services: Scoping Down Access, choosing Amazon Timestream in step 5.

Create a new QuickSight data source connection for Timestream

> **Note**
>
> The connection between Amazon QuickSight and Amazon Timestream is encrypted in transit using SSL (TLS 1.2). You cannot create an unencrypted connection.

1. Ensure you have configured the appropriate permissions for Amazon QuickSight to access Amazon Timestream, as described in Accessing Amazon Timestream from QuickSight.
2. Begin by creating a new dataset. Choose **Datasets** from the navigation pane, then choose **New Dataset**.
3. Select the Timestream data source card.
4. For **Data source name**, enter a name for your Timestream data source connection, for example `US Timestream Data`.

   > **Note**
   >
   > Because you can create many datasets from a connection to Timestream, it's best to keep the name simple.

5. Choose **Validate connection** to check that you can successfully connect to Timestream.

   > **Note**
   >
   > **Validate connection** only validates that you can connect. However, it doesn't validate a specific table or query.

6. Choose **Create data source** to proceed.

7. For **Database**, choose **Select...** to view the list of available options. Choose the one you want to use.
8. Choose **Select** to continue.
9. Choose one of the following:
   - To import your data into QuickSight's in-memory engine (called SPICE), choose **Import to SPICE for quicker analytics**.
   - To allow QuickSight to run a query against your data each time you refresh the dataset or use the analysis or dashboard, choose **Directly query your data**.
10. Choose **Edit/Preview** and then **Save** to save your dataset and close it.

Edit permissions for the QuickSight data source connection for Timestream

The following procedure describes how to view, add, and revoke permissions for other QuickSight users so that they can access the same Timestream data source. The people need to be active users in QuickSight before you can add them.

> **Note**
>
> In QuickSight, data sources have two permissions levels: user and owner.
>
> - Choose *user* to allow read access.
> - Choose *owner* to allow that user to edit, share, or delete this QuickSight data source.

1. Ensure you have configured the appropriate permissions for Amazon QuickSight to access Amazon Timestream, as described in Accessing Amazon Timestream from QuickSight.
2. Choose **Datasets** at left, then scroll down to find the data source card for your Timestream connection. For example `US Timestream Data`.
3. Choose the `Timestream` data source card.
4. Choose `Share data source`. A list of current permissions displays.
5. (Optional) To edit permissions, you can choose `user` or `owner`.
6. (Optional) To revoke permissions, choose `Revoke access`. People you revoke can't create new datasets from this data source. However, their existing datasets will still have access to this data source.

7. To add permissions, choose `Invite users`, then follow these steps to add a user:
   a. Add people to allow them to use the same data source.
   b. For each, choose the `Permission` that you want to apply.
8. When you are finished, choose `Close`.

Create a new QuickSight dataset for Timestream

1. Ensure you have configured the appropriate permissions for Amazon QuickSight to access Amazon Timestream, as described in Accessing Amazon Timestream from QuickSight.
2. Choose **Datasets** at left, then scroll down to find the data source card for your Timestream connection. If you have many data sources, you can use the search bar at the top of the page to find it with a partial match on the name.
3. Choose the **Timestream** data source card. Then choose **Create data set**.
4. For **Database**, choose **Select** to view the list of available options. Choose the database that you want to use.
5. For **Tables**, choose the table that you want to use.
6. Choose **Edit/Preview**.
7. (Optional) To add more data, choose **Add data** at top right.
   a. Choose **Switch data source**, and choose a different data source.
   b. Follow the UI prompts to finish adding data.
   c. After adding new data to the same dataset, choose **Configure this join** (the two red dots). Set up a join for each additional table.
   d. If you want to add calculated fields, choose **Add calculated field**.
   e. To use Sagemaker, choose **Augment with SageMaker**. This option is only available in QuickSight Enterprise edition.
   f. Uncheck any fields you want to omit.
   g. Update any data types you want to change.
8. When you are done, choose **Save** to save and close the dataset.

Create a new analysis for Timestream

1. Ensure you have configured the appropriate permissions for Amazon QuickSight to access Amazon Timestream, as described in Accessing Amazon Timestream from QuickSight.
2. Choose **Analyses** at left.
3. Choose one of the following:
   - To create a new analysis, choose **New analysis** at right.
   - To add the Timestream dataset to an existing analysis, open the analysis you want to edit. Choose the pencil icon near at top left, then **Add data set**.
4. Start the first data visualization by choosing fields on the left.
5. For more information, see Working with Analyses - Amazon QuickSight

Video tutorial

This video explains how Quick works with Timestream.

Amazon SageMaker AI

You can use Amazon SageMaker Notebooks to integrate your machine learning models with Amazon Timestream. To help you get started, we have created a sample SageMaker Notebook that processes data from Timestream. The data is inserted into Timestream from a multi-threaded Python application continuously sending data. The source code for the sample SageMaker Notebook and the sample Python application are available in GitHub.

1. Create a database and table following the instructions described in Create a database and Create a table.
2. Clone the GitHub repository for the multi-threaded Python sample application following the instructions from GitHub.
3. Clone the GitHub repository for the sample Timestream SageMaker Notebook following the instructions from GitHub.
4. Run the application for continuously ingesting data into Timestream following the instructions in the README.
5. Follow the instructions to create an Amazon S3 bucket for Amazon SageMaker as described here.

6. Create an Amazon SageMaker instance with latest boto3 installed: In addition to the instructions described here, follow the steps below:

   a. On the **Create notebook** instance page, click on **Additional Configuration**

   b. Click on **Lifecycle configuration - *optional*** and select **Create a new lifecycle configuration**

   c. On the *Create lifecycle configuration* wizard box, do the following:

      i. Fill in a desired name to the configuration, e.g. `on-start`

      ii. In Start Notebook script, copy-paste the script content from Github

      iii. Replace `PACKAGE=scipy` with `PACKAGE=boto3` in the pasted script.

7. Click on **Create configuration**

8. Go to the IAM service in the AWS Management Console and find the newly created SageMaker execution role for the notebook instance.

9. Attach the IAM policy for `AmazonTimestreamFullAccess` to the execution role.

> **Note**
>
> The `AmazonTimestreamFullAccess` IAM policy is not restricted to specific resources and is unsuitable for production use. For a production system, consider using policies that restrict access to specific resources.

10. When the status of the notebook instance is **InService**, choose **Open Jupyter** to launch a SageMaker Notebook for the instance

11. Upload the files `timestreamquery.py` and `Timestream_SageMaker_Demo.ipynb` into the Notebook by selecting the **Upload** button

12. Choose `Timestream_SageMaker_Demo.ipynb`

> **Note**
>
> If you see a pop up with **Kernel not found**, choose **conda_python3** and click **Set Kernel**.

13. Modify `DB_NAME`, `TABLE_NAME`, `bucket`, and `ENDPOINT` to match the database name, table name, S3 bucket name, and region for the training models.

14. Choose the **play** icon to run the individual cells

15. When you get to the cell `Leverage Timestream to find hosts with average CPU utilization across the fleet`, ensure that the output returns at least 2 host names.

> **Note**
>
> If there are less than 2 host names in the output, you may need to rerun the sample Python application ingesting data into Timestream with a larger number of threads and host-scale.

16. When you get to the cell `Train a Random Cut Forest (RCF) model using the CPU utilization history`, change the `train_instance_type` based on the resource requirements for your training job
17. When you get to the cell `Deploy the model for inference`, change the `instance_type` based on the resource requirements for your inference job

> **Note**
>
> It may take a few minutes to train the model. When the training is complete, you will see the message **Completed - Training job completed** in the output of the cell.

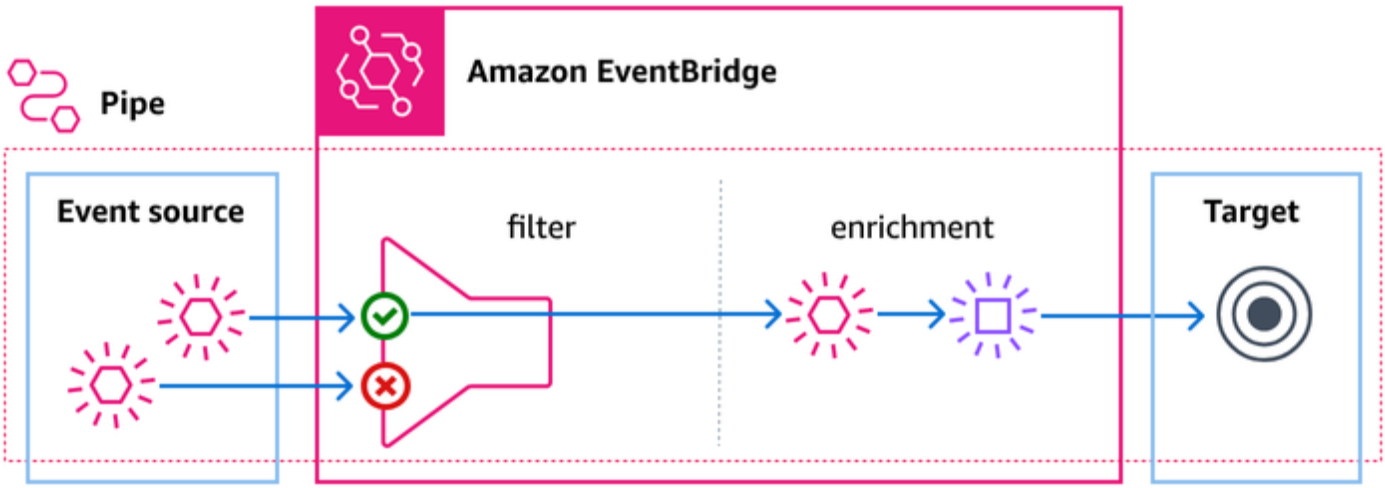
18. Run the cell `Stop and delete the endpoint` to clean up resources. You can also stop and delete the instance from the SageMaker console

Amazon SQS

Using EventBridge Pipes to send Amazon SQS data to Timestream

You can use EventBridge Pipes to send data from a Amazon SQS queue to a Amazon Timestream for LiveAnalytics table.

Pipes are intended for point-to-point integrations between supported sources and targets, with support for advanced transformations and enrichment. Pipes reduce the need for specialized knowledge and integration code when developing event-driven architectures. To set up a pipe, you choose the source, add optional filtering, define optional enrichment, and choose the target for the event data.



For more information on EventBridge Pipes, see EventBridge Pipes in the *EventBridge User Guide*. For information on configuring a pipe to deliver events to a Amazon Timestream for LiveAnalytics table, see EventBridge Pipes target specifics.

Using DBeaver to work with Amazon Timestream

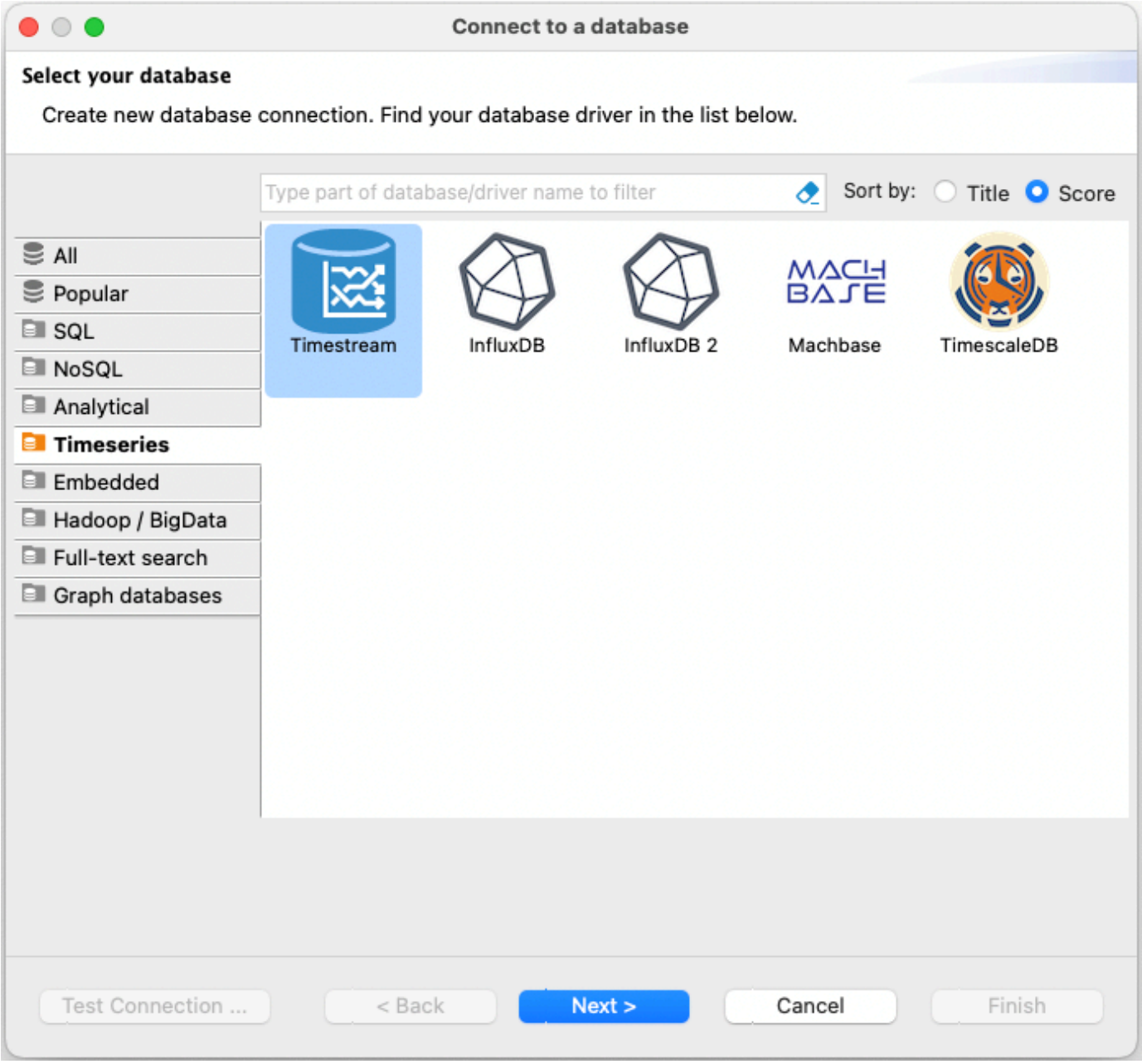
DBeaver is a free universal SQL client that can be used to manage any database that has a JDBC driver. It is widely used among developers and database administrators because of its robust data viewing, editing, and management capabilities.

Using DBeaver's cloud connectivity options, you can connect DBeaver to Amazon Timestream natively. DBeaver provides a comprehensive and intuitive interface to work with time series data directly from within a DBeaver application. Using your credentials, it also gives you full access to any queries that you could execute from another query interface. It even lets you create graphs for better understanding and visualization of query results.

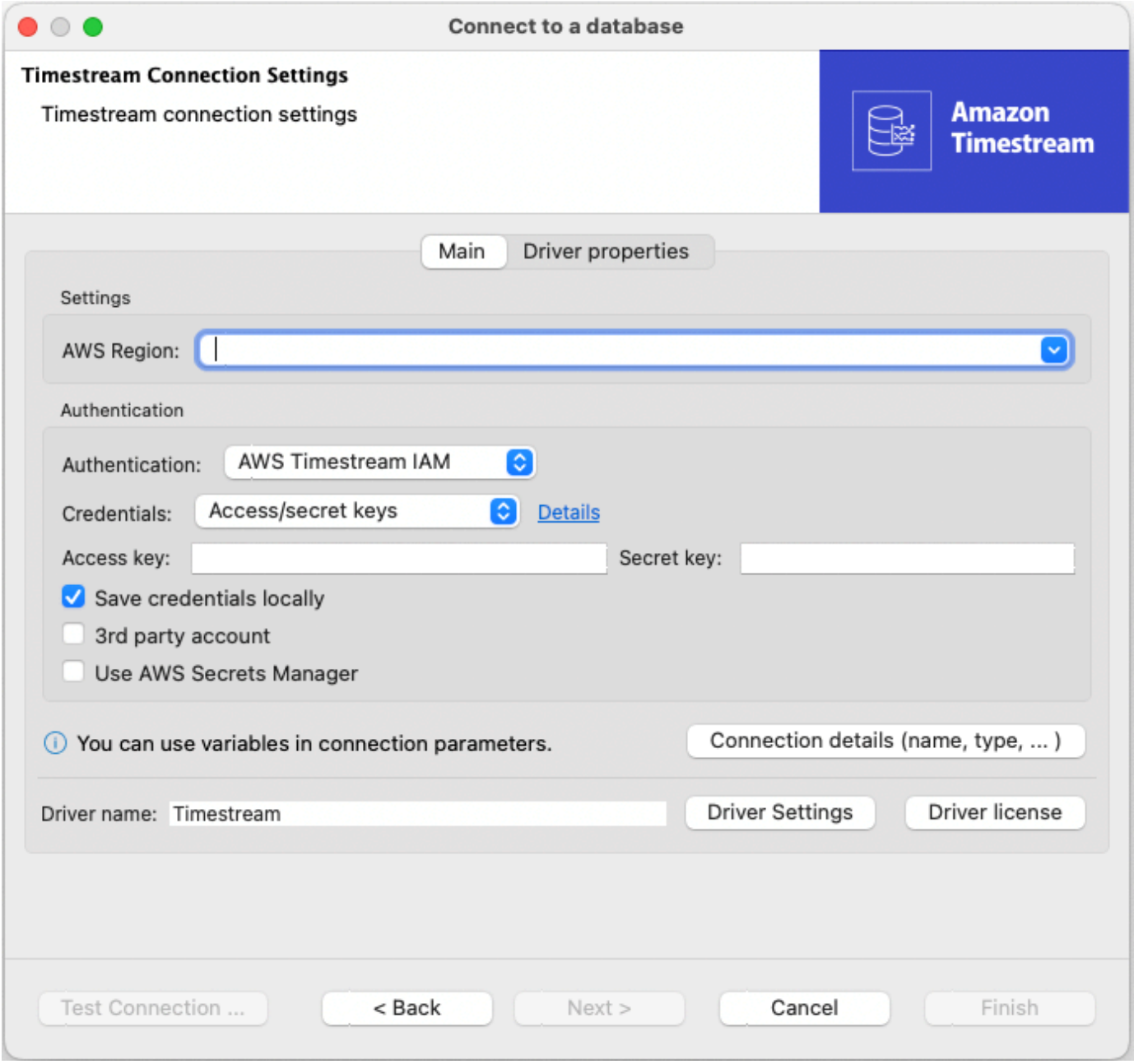
Setting up DBeaver to work with Timestream

Take the following steps to set up DBeaver to work with Timestream:

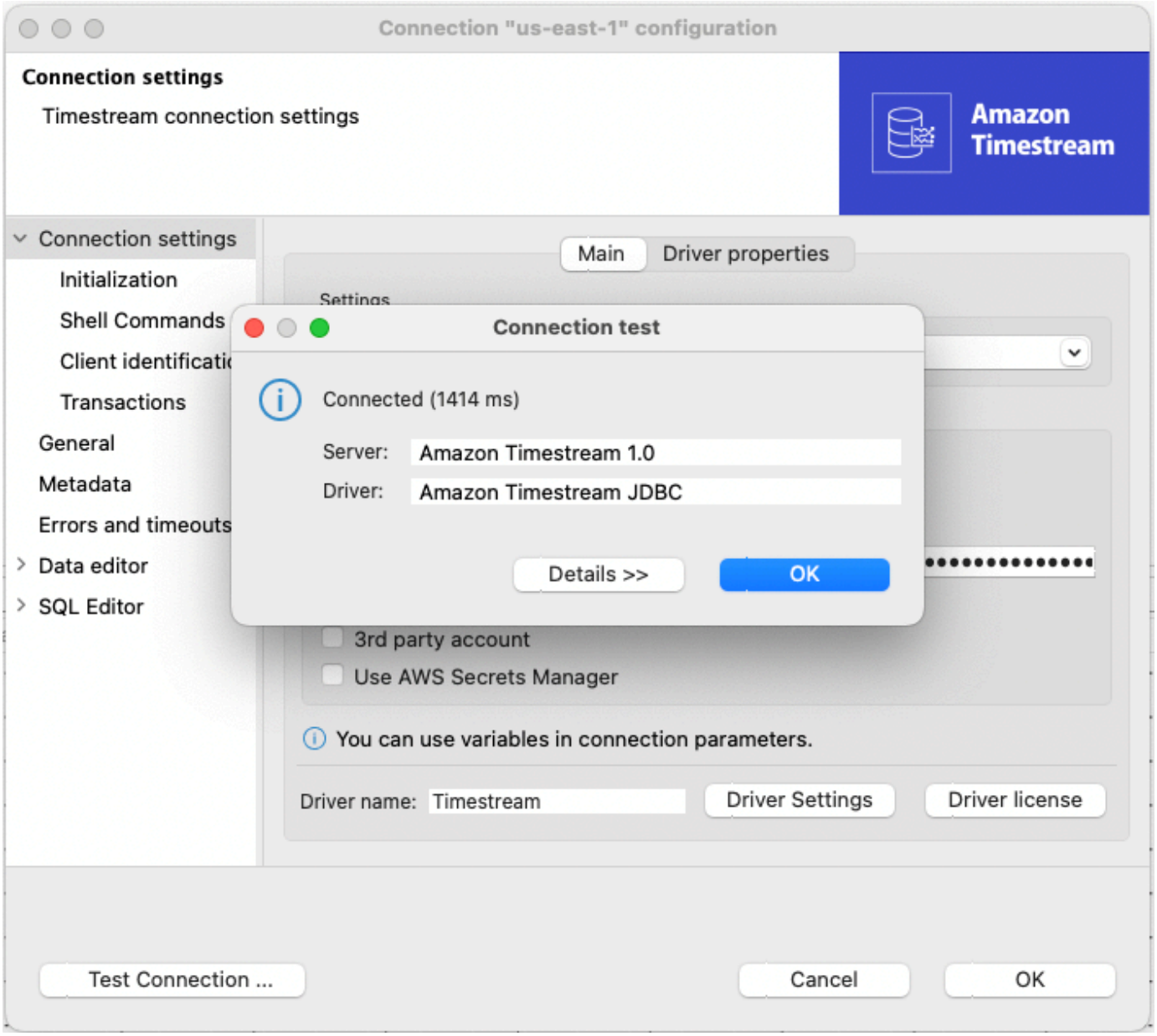
1. Download and install DBeaver on your local machine.
2. Launch DBeaver, navigate to the database selection area, choose **Timeseries** in the left pane, and then select the **Timestream** icon in the right pane:



3. In the **Timestream Connection Settings** window, enter all the information necessary to connect to your Amazon Timestream database. Please ensure that the user keys you enter have the permissions necessary to access your Timestream database. Also, be sure to keep the information and keys you input into DBeaver safe and private, as with any sensitive information.



4. Test the connection to ensure that everything is set up correctly:



5. If the connection test is successful, you can now interact with your Amazon Timestream database just as you would with any other database in DBeaver. For example, you can navigate to the SQL editor or to the ER Diagram view to run queries:

6. DBeaver also provides powerful data visualization tools. To use them, run your query, then select the graph icon to visualize the result set. The graphing tool can help you better understand data trends over time.

Pairing Amazon Timestream with DBeaver creates an effective environment for managing time series data. You can integrate it seamlessly into your existing workflow to enhance productivity and efficiency.

Grafana

You can visualize your time series data and create alerts using Grafana. To help you get started with data visualization, we have created a sample dashboard in Grafana that visualizes data sent to Timestream from a Python application and a video tutorial that describes the setup.

Topics

- Sample application
- Video tutorial

Sample application

1. Create a database and a table in Timestream following the instructions described in Create a database for more information.

   > **Note**
   >
   > The default database name and table name for the Grafana dashboard are set to grafanaDB and grafanaTable respectively. Use these names to minimize setup.

2. Install Python 3.7 or higher.
3. Install and configure the Timestream Python SDK.s
4. Clone the GitHub repository for the multi-thread Python application continuously ingesting data into Timestream following the instructions from GitHub.
5. Run the application for continuously ingesting data into Timestream following the instructions in the README.
6. Complete Learn how to create and use Amazon Managed Grafana resources or complete Install Grafana.
7. If installing Grafana instead of using Amazon Managed Grafana, complete Installing Amazon Timestream on Grafana Cloud.
8. Open the Grafana dashboard using a browser of your choice. If you've locally installed Grafana, you can follow the instructions described in the Grafana documentation to log in.

9. After launching Grafana, go to Datasources, click on Add Datasource, search for Timestream, and select the Timestream datasource.
10. Configure the Auth Provider and the region and click Save and Test.
11. Set the default macros.
    a. Set $__database to the name of your Timestream database (e.g. grafanaDB).
    b. Set $__table to the name of your Timestream table (e.g. grafanaTable).
    c. Set $__measure to the most commonly used measure from the table.
12. Click Save and Test.
13. Click on the Dashboards tab.
14. Click on Import to import the dashboard.
15. Double click the Sample Application Dashboard.
16. Click on the dashboard settings.
17. Select Variables.
18. Change dbName and tableName to match the names of the Timestream database and table.
19. Click Save.
20. Refresh the dashboard.
21. To create alerts, follow the instructions described in the Grafana documentation to Configure Grafana-managed alert rules.
22. To troubleshoot alerts, follow the instructions described in the Grafana documentation for Troubleshooting.
23. For additional information, see the Grafana documentation.

Video tutorial

This video explains how Grafana works with Timestream.

Using SquaredUp to work with Amazon Timestream

SquaredUp is an observability platform that integrates with Amazon Timestream. You can use SquaredUp's intuitive dashboard designer to visualize, analyze, and monitor your time-series data. Dashboards can be shared publicly or privately, and notification channels can be created to alert you when the health state of a monitor changes.

Using SquaredUp with Amazon Timestream

1. Sign up for SquaredUp and get started for free.
2. Add an AWS data source.
3. Create a dashboard tile that uses the Timestream Query data stream.
4. Optionally, enable monitoring for the tile, create a notification channel, or share the dashboard publicly or privately.
5. Optionally create other tiles to see your Timestream data alongside data from your other monitoring and observability tools.

Open source Telegraf

You can use the Timestream for LiveAnalytics output plugin for Telegraf to write metrics into Timestream for LiveAnalytics directly from open source Telegraf.

This section provides an explanation of how to install Telegraf with the Timestream for LiveAnalytics output plugin, how to run Telegraf with the Timestream for LiveAnalytics output plugin, and how open source Telegraf works with Timestream for LiveAnalytics.

Topics

- Installing Telegraf with the Timestream for LiveAnalytics output plugin
- Running Telegraf with the Timestream for LiveAnalytics output plugin
- Mapping Telegraf/InfluxDB metrics to the Timestream for LiveAnalytics model

Installing Telegraf with the Timestream for LiveAnalytics output plugin

As of version 1.16, the Timestream for LiveAnalytics output plugin is available in the official Telegraf release. To install the output plugin on most major operating systems, follow the steps outlined in the InfluxData Telegraf Documentation. To install on the Amazon Linux 2 OS, follow the instructions below.

Installing Telegraf with the Timestream for LiveAnalytics output plugin on Amazon Linux 2

To install Telegraf with the Timestream Output Plugin on Amazon Linux 2, perform the following steps.

1. Install Telegraf using the `yum` package manager.

```
cat <<EOF | sudo tee /etc/yum.repos.d/influxdb.repo
[influxdb]
name = InfluxDB Repository - RHEL \$releasever
baseurl = https://repos.influxdata.com/rhel/\$releasever/\$basearch/stable
enabled = 1
gpgcheck = 1
gpgkey = https://repos.influxdata.com/influxdb.key
EOF
```

2. Run the following command.

```
sudo sed -i "s/\$releasever/$(rpm -E %{rhel})/g" /etc/yum.repos.d/influxdb.repo
```

3. Install and start Telegraf.

```
sudo yum install telegraf
sudo service telegraf start
```

Running Telegraf with the Timestream for LiveAnalytics output plugin

You can follow the instructions below to run Telegraf with the Timestream for LiveAnalytics plugin.

1. Generate an example configuration using Telegraf.

```
telegraf --section-filter agent:inputs:outputs --input-filter cpu:mem --output-
filter timestream config > example.config
```

2. Create a database in Timestream using the management console, CLI, or SDKs.
3. In the `example.config` file, add your database name by editing the following key under the `[[outputs.timestream]]` section.

```
database_name = "yourDatabaseNameHere"
```

4. By default, Telegraf will create a table. If you wish create a table manually, set `create_table_if_not_exists` to `false` and follow the instructions to create a table using the management console, CLI, or SDKs.
5. In the *example.config* file, configure credentials under the `[[outputs.timestream]]` section. The credentials should allow the following operations.

```
timestream:DescribeEndpoints
timestream:WriteRecords
```

> **Note**
>
> If you leave `create_table_if_not_exists` set to `true`, include:
>
> ```
> timestream:CreateTable
> ```

> **Note**
>
> If you set `describe_database_on_start` to `true`, include the following.
>
> ```
> timestream:DescribeDatabase
> ```

6. You can edit the rest of the configuration according to your preferences.
7. When you have finished editing the config file, run Telegraf with the following.

```
./telegraf --config example.config
```

8. Metrics should appear within a few seconds, depending on your agent configuration. You should also see the new tables, *cpu* and *mem*, in the Timestream console.

Mapping Telegraf/InfluxDB metrics to the Timestream for LiveAnalytics model

When writing data from Telegraf to Timestream for LiveAnalytics, the data is mapped as follows.

- The timestamp is written as the time field.
- Tags are written as dimensions.
- Fields are written as measures.
- Measurements are mostly written as table names (more on this below).

The Timestream for LiveAnalytics output plugin for Telegraf offers multiple options for organizing and storing data in Timestream for LiveAnalytics. This can be described with an example which begins with the data in line protocol format.

```
weather,location=us-midwest,season=summer temperature=82,humidity=71
1465839830100400200 airquality,location=us-west no2=5,pm25=16
1465839830100400200
```

The following describes the data.

- The measurement names are `weather` and `airquality`.
- The tags are `location` and `season`.
- The fields are `temperature`, `humidity`, no2, and pm25.

Topics

- Storing the data in multiple tables
- Storing the data in a single table

Storing the data in multiple tables

You can choose to create a separate table per measurement and store each field in a separate row per table.

The configuration is `mapping_mode = "multi-table"`.

- The Timestream for LiveAnalytics adapter will create two tables, namely, `weather` and `airquality`.
- Each table row will contain a single field only.

The resulting Timestream for LiveAnalytics tables, `weather` and `airquality`, will look like this.

`weather`

time	location	season	measure_name	measure_value::bigint
2016-06-13 17:43:50	us-midwest	summer	temperature	82

time	location	season	measure_name	measure_value::bigint
2016-06-13 17:43:50	us-midwest	summer	humidity	71

`airquality`

time	location	measure_name	measure_value::bigint
2016-06-13 17:43:50	us-midwest	no2	5
2016-06-13 17:43:50	us-midwest	pm25	16

Storing the data in a single table

You can choose to store all the measurements in a single table and store each field in a separate table row.

The configuration is `mapping_mode = "single-table"`. There are two addition configurations when using `single-table`, `single_table_name` and `single_table_dimension_name_for_telegraf_measurement_name`.

- The Timestream for LiveAnalytics output plugin will create a single table with name *`<single_table_name>`* which includes a *`<single_table_dimension_name_for_telegraf_measurement_name>`* column.
- The table may contain multiple fields in a single table row.

The resulting Timestream for LiveAnalytics table will look like this.

weather

time	location	season	*<single_table_dimension_name_for_telegraf_measurement_name>*	measure_name	measure_value::bigint
2016-06-13 17:43:50	us-midwest	summer	weather	temperature	82
2016-06-13 17:43:50	us-midwest	summer	weather	humidity	71
2016-06-13 17:43:50	us-midwest	summer	airquality	no2	5
2016-06-13 17:43:50	us-midwest	summer	weather	pm25	16

JDBC

You can use a Java Database Connectivity (JDBC) connection to connect Timestream for LiveAnalytics to your business intelligence tools and other applications, such as SQL Workbench. The Timestream for LiveAnalytics JDBC driver currently supports SSO with Okta and Microsoft Azure AD.

Topics

- Configuring the JDBC driver for Timestream for LiveAnalytics
- Connection properties
- JDBC URL examples
- Setting up Timestream for LiveAnalytics JDBC single sign-on authentication with Okta
- Setting up Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD

Configuring the JDBC driver for Timestream for LiveAnalytics

Follow the steps below to configure the JDBC driver.

Topics

- Timestream for LiveAnalytics JDBC driver JARs
- Timestream for LiveAnalytics JDBC driver class and URL format
- Sample application

Timestream for LiveAnalytics JDBC driver JARs

You can obtain the Timestream for LiveAnalytics JDBC driver via direct download or by adding the driver as a Maven dependency.

- *As a direct download:*. To directly download the Timestream for LiveAnalytics JDBC driver, complete the following steps:

  1. Navigate to https://github.com/awslabs/amazon-timestream-driver-jdbc/releases
  2. You can use `amazon-timestream-jdbc-1.0.1-shaded.jar` directly with your business intelligence tools and applications
  3. Download `amazon-timestream-jdbc-1.0.1-javadoc.jar` to a directory of your choice.
  4. In the directory where you have downloaded `amazon-timestream-jdbc-1.0.1-javadoc.jar`, run the following command to extract the Javadoc HTML files:

     ```
     jar -xvf amazon-timestream-jdbc-1.0.1-javadoc.jar
     ```

- *As a Maven dependency:* To add the Timestream for LiveAnalytics JDBC driver as a Maven dependency, complete the following steps:

  1. Navigate to and open your application's `pom.xml` file in an editor of your choice.
  2. Add the JDBC driver as a dependency into your application's `pom.xml` file:

     ```
     <!-- https://mvnrepository.com/artifact/software.amazon.timestream/amazon-
     timestream-jdbc -->
     <dependency>
         <groupId>software.amazon.timestream</groupId>
         <artifactId>amazon-timestream-jdbc</artifactId>
     ```

```
    <version>1.0.1</version>
</dependency>
```

Timestream for LiveAnalytics JDBC driver class and URL format

The driver class for Timestream for LiveAnalytics JDBC driver is:

```
software.amazon.timestream.jdbc.TimestreamDriver
```

The Timestream JDBC driver requires the following JDBC URL format:

```
jdbc:timestream:
```

To specify database properties through the JDBC URL, use the following URL format:

```
jdbc:timestream://
```

Sample application

To help you get started with using Timestream for LiveAnalytics with JDBC, we've created a fully functional sample application in GitHub.

1. Create a database with sample data following the instructions described here.
2. Clone the GitHub repository for the sample application for JDBC following the instructions from GitHub.
3. Follow the instructions in the README to get started with the sample application.

Connection properties

The Timestream for LiveAnalytics JDBC driver supports the following options:

Topics

- Basic authentication options
- Standard client info option
- Driver configuration option

- SDK option
- Endpoint configuration option
- Credential provider options
- SAML-based authentication options for Okta
- SAML-based authentication options for Azure AD

> **Note**
>
> If none of the properties are provided, the Timestream for LiveAnalytics JDBC driver will use the default credentials chain to load the credentials.

> **Note**
>
> All property keys are case-sensitive.

Basic authentication options

The following table describes the available Basic Authentication options.

Option	Description	Default
AccessKeyId	The AWS user access key id.	NONE
SecretAccessKey	The AWS user secret access key.	NONE
SessionToken	The temporary session token required to access a database with multi-factor authentic ation (MFA) enabled.	NONE

Standard client info option

The following table describes the Standard Client Info Option.

Option	Description	Default
ApplicationName	The name of the applicati on currently utilizing the connection. `ApplicationName` is used for debugging purposes and will not be communicated to the Timestream for LiveAnalytics service.	The application name detected by the driver.

Driver configuration option

The following table describes the Driver Configuration Option.

Option	Description	Default
EnableMetaDataPreparedStatement	Enables Timestream for LiveAnalytics JDBC driver to return metadata for `PreparedStatements`, but this will incur an additional cost with Timestream for LiveAnalytics when retrieving the metadata.	FALSE
Region	The database's region.	us-east-1

SDK option

The following table describes the SDK Option.

Option	Description	Default
RequestTimeout	The time in milliseconds the AWS SDK will wait for a query	0

Option	Description	Default
	request before timing out. Non-positive value disables request timeout.	
SocketTimeout	The time in milliseconds the AWS SDK will wait for data to be transferred over an open connection before timing out. Value must be non-negative. A value of `0` disables socket timeout.	50000
MaxRetryCountClient	The maximum number of retry attempts for retryable errors with 5XX error codes in the SDK. The value must be non-negative.	NONE
MaxConnections	The maximum number of allowed concurrently opened HTTP connections to the Timestream for LiveAnalytics service. The value must be positive.	50

Endpoint configuration option

The following table describes the Endpoint Configuration Option.

Option	Description	Default
Endpoint	The endpoint for the Timestream for LiveAnalytics service.	NONE

Credential provider options

The following table describes the available Credential Provider options.

Option	Description	Default
AwsCredentialsProviderClass	One of `PropertiesFileCredentialsProvider` or `InstanceProfileCredentialsProvider` to use for authentication.	NONE
CustomCredentialsFilePath	The path to a properties file containing AWS security credentials `accessKey` and `secretKey`. This is only required if `AwsCredentialsProviderClass` is specified as `PropertiesFileCredentialsProvider`.	NONE

SAML-based authentication options for Okta

The following table describes the available SAML-based authentication options for Okta.

Option	Description	Default
IdpName	The Identity Provider (Idp) name to use for SAML-based authentication. One of `Okta` or `AzureAD`.	NONE
IdpHost	The host name of the specified Idp.	NONE

Option	Description	Default
IdpUserName	The user name for the specified Idp account.	NONE
IdpPassword	The password for the specified Idp account.	NONE
OktaApplicationID	The unique Okta-provided ID associated with the Timestream for LiveAnalytics application. `AppId` can be found in the `entityID` field provided in the application metadata. Consider the following example: `entityID = http://www.okta.com//IdpAppID`	NONE
RoleARN	The Amazon Resource Name (ARN) of the role that the caller is assuming.	NONE
IdpARN	The Amazon Resource Name (ARN) of the SAML provider in IAM that describes the Idp.	NONE

SAML-based authentication options for Azure AD

The following table describes the available SAML-based authentication options for Azure AD.

Option	Description	Default
IdpName	The Identity Provider (Idp) name to use for SAML-based authentication. One of `Okta` or `AzureAD` .	NONE

Option	Description	Default
IdpHost	The host name of the specified Idp.	NONE
IdpUserName	The user name for the specified Idp account.	NONE
IdpPassword	The password for the specified Idp account.	NONE
AADApplicationID	The unique id of the registere d application on Azure AD.	NONE
AADClientSecret	The client secret associated with the registered applicati on on Azure AD used to authorize fetching tokens.	NONE
AADTenant	The Azure AD Tenant ID.	NONE
IdpARN	The Amazon Resource Name (ARN) of the SAML provider in IAM that describes the Idp.	NONE

JDBC URL examples

This section describes how to create a JDBC connection URL, and provides examples. To specify the optional connection properties, use the following URL format:

```
jdbc:timestream://PropertyName1=value1;PropertyName2=value2...
```

> **Note**
>
> All connection properties are optional. All property keys are case-sensitive.

Below are some examples of JDBC connection URLs.

Example with basic authentication options and region:

```
jdbc:timestream://
AccessKeyId=<myAccessKeyId>;SecretAccessKey=<mySecretAccessKey>;SessionToken=<mySessionToken
east-1
```

Example with client info, region and SDK options:

```
jdbc:timestream://ApplicationName=MyApp;Region=us-
east-1;MaxRetryCountClient=10;MaxConnections=5000;RequestTimeout=20000
```

Connect using the default credential provider chain with AWS credential set in environment variables:

```
jdbc:timestream
```

Connect using the default credential provider chain with AWS credential set in the connection URL:

```
jdbc:timestream://
AccessKeyId=<myAccessKeyId>;SecretAccessKey=<mySecretAccessKey>;SessionToken=<mySessionToken
```

Connect using the PropertiesFileCredentialsProvider as the authentication method:

```
jdbc:timestream://
AwsCredentialsProviderClass=PropertiesFileCredentialsProvider;CustomCredentialsFilePath=<pat
 to properties file>
```

Connect using the InstanceProfileCredentialsProvider as the authentication method:

```
jdbc:timestream://AwsCredentialsProviderClass=InstanceProfileCredentialsProvider
```

Connect using the Okta credentials as the authentication method:

```
jdbc:timestream://
IdpName=Okta;IdpHost=<host>;IdpUserName=<name>;IdpPassword=<password>;OktaApplicationID=<id>
```

Connect using the Azure AD credentials as the authentication method:

```
jdbc:timestream://
IdpName=AzureAD;IdpUserName=<name>;IdpPassword=<password>;AADApplicationID=<id>;AADClientSec
```

Connect with a specific endpoint:

```
jdbc:timestream://Endpoint=abc.us-east-1.amazonaws.com;Region=us-east-1
```

Setting up Timestream for LiveAnalytics JDBC single sign-on authentication with Okta

Timestream for LiveAnalytics supports Timestream for LiveAnalytics JDBC single sign-on authentication with Okta. To use Timestream for LiveAnalytics JDBC single sign-on authentication with Okta, complete each of the sections listed below.

Topics

- Prerequisites
- AWS account federation in Okta
- Setting up Okta for SAML

Prerequisites

Ensure that you have met the following prerequisites before using the Timestream for LiveAnalytics JDBC single sign-on authentication with Okta:

- Admin permissions in AWS to create the identity provider and the roles.
- An Okta account (Go to https://www.okta.com/login/ to create an account).
- Access to Amazon Timestream for LiveAnalytics.

Now that you have completed the Prerequisites, you may proceed to AWS account federation in Okta.

AWS account federation in Okta

The Timestream for LiveAnalytics JDBC driver supports AWS Account Federation in Okta. To set up AWS Account Federation in Okta, complete the following steps:

1. Sign in to the Okta Admin dashboard using the following URL:

```
https://<company-domain-name>-admin.okta.com/admin/apps/active
```

> **Note**
>
> Replace **<company-domain-name>** with your domain name.

2. Upon successful sign-in, choose **Add Application** and search for **AWS Account Federation**.
3. Choose **Add**
4. Change the Login URL to the appropriate URL.
5. Choose **Next**
6. Choose **SAML 2.0** As the **Sign-On** method
7. Choose **Identity Provider metadata** to open the metadata XML file. Save the file locally.
8. Leave all other configuration options blank.
9. Choose **Done**

Now that you have completed AWS Account Federation in Okta, you may proceed to Setting up Okta for SAML.

Setting up Okta for SAML

1. Choose the **Sign On** tab. Choose the **View**.
2. Choose the **Setup Instructions** button in the **Settings** section.

Finding the Okta metadata document

1. To find the document, go to:

```
https://<domain>-admin.okta.com/admin/apps/active
```

> **Note**
>
> <domain> is your unique domain name for your Okta account.

2. Choose the **AWS Account Federation** application
3. Choose the **Sign On** tab

Setting up Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD

Timestream for LiveAnalytics supports Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD. To use Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD, complete each of the sections listed below.

Topics

- Prerequisites
- Setting up Azure AD
- Setting up IAM Identity Provider and roles in AWS

Prerequisites

Ensure that you have met the following prerequisites before using the Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD:

- Admin permissions in AWS to create the identity provider and the roles.
- An Azure Active Directory account (Go to https://azure.microsoft.com/en-ca/services/active-directory/ to create an account)
- Access to Amazon Timestream for LiveAnalytics.

Setting up Azure AD

1. Sign in to Azure Portal
2. Choose **Azure Active Directory** in the list of Azure services. This will redirect to the Default Directory page.
3. Choose **Enterprise Applications** under the **Manage** section on the sidebar
4. Choose **+ New application**.
5. Find and select **Amazon Web Services**.
6. Choose **Single Sign-On** under the **Manage** section in the sidebar
7. Choose **SAML** as the single sign-on method
8. In the Basic SAML Configuration section, enter the following URL for both the Identifier and the Reply URL:

```
https://signin.aws.amazon.com/saml
```

9. Choose **Save**
10. Download the Federation Metadata XML in the SAML Signing Certificate section. This will be used when creating the IAM Identity Provider later
11. Return to the Default Directory page and choose **App registrations** under **Manage**.
12. Choose **Timestream for LiveAnalytics** from the **All Applications** section. The page will be redirected to the application's Overview page

> **Note**
>
> Note the Application (client) ID and the Directory (tenant) ID. These values are required for when creating a connection.

13. Choose **Certificates and Secrets**
14. Under **Client secrets**, create a new client secret with **+ New client secret**.

> **Note**
>
> Note the generated client secret, as this is required when creating a connection to Timestream for LiveAnalytics.

15. On the sidebar under **Manage**, select **API permissions**
16. In the **Configured permissions**, use **Add a permission** to grant Azure AD permission to sign in to Timestream for LiveAnalytics. Choose **Microsoft Graph** on the Request API permissions page.
17. Choose **Delegated permissions** and select the **User.Read** permission
18. Choose **Add permissions**
19. Choose **Grant admin consent for Default Directory**

Setting up IAM Identity Provider and roles in AWS

Complete each section below to set up IAM for Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD:

Topics

- Create a SAML Identity Provider

- Create an IAM role
- Create an IAM policy
- Provisioning

Create a SAML Identity Provider

To create a SAML Identity Provider for the Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD, complete the following steps:

1. Sign in to the AWS Management Console
2. Choose **Services** and select **IAM** under Security, Identity, & Compliance
3. Choose **Identity providers** under Access management
4. Choose **Create Provider** and choose **SAML** as the provider type. Enter the **Provider Name**. This example will use AzureADProvider.
5. Upload the previously downloaded Federation Metadata XML file
6. Choose **Next**, then choose **Create**.
7. Upon completion, the page will be redirected back to the Identity providers page

Create an IAM role

To create an IAM role for the Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD, complete the following steps:

1. On the sidebar select **Roles** under Access management
2. Choose Create role
3. Choose **SAML 2.0 federation** as the trusted entity
4. Choose the **Azure AD provider**
5. Choose **Allow programmatic and AWS Management Console access**
6. Choose **Next: Permissions**
7. Attach permissions policies or continue to Next:Tags
8. Add optional tags or continue to Next:Review
9. Enter a Role name. This example will use AzureSAMLRole

10Provide a role description

11. Choose **Create Role** to complete

Create an IAM policy

To create an IAM policy for the Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD complete the following steps:

1. On the sidebar, choose **Policies** under Access management
2. Choose **Create policy** and select the **JSON** tab
3. Add the following policy

   JSON

```
{
"Version":"2012-10-17",
"Statement": [
    {
          "Effect": "Allow",
          "Action": [
                 "iam:ListRoles",
                 "iam:ListAccountAliases"
           ],
           "Resource": "*"
      }
]
}
```

4. Choose **Create policy**
5. Enter a policy name. This example will use TimestreamAccessPolicy.
6. Choose **Create Policy**
7. On the sidebar, choose **Roles** under Access management.
8. Choose the previously created **Azure AD role** and choose **Attach policies** under Permissions.
9. Select the previously created access policy.

Provisioning

To provision the identity provider for Timestream for LiveAnalytics JDBC single sign-on authentication with Microsoft Azure AD, complete the following steps:

1. Go back to Azure Portal
2. Choose **Azure Active Directory** in the list of Azure services. This will redirect to the Default Directory page
3. Choose **Enterprise Applications** under the Manage section on the sidebar
4. Choose **Provisioning**
5. Choose **Automatic mode** for the Provisioning Method
6. Under Admin Credentials, enter your **AwsAccessKeyID** for clientsecret, and **SecretAccessKey** for Secret Token
7. Set the **Provisioning Status** to **On**
8. Choose **save**. This allows Azure AD to load the necessary IAM Roles
9. Once the Current cycle status is completed, choose **Users and groups** on the sidebar
10. Choose **+ Add user**
11. Choose the Azure AD user to provide access to Timestream for LiveAnalytics
12. Choose the IAM Azure AD role and the corresponding Azure Identity Provider created in AWS
13. Choose **Assign**

ODBC

The open-source ODBC driver for Amazon Timestream for LiveAnalytics provides an SQL-relational interface to Timestream for LiveAnalytics for developers and enables connectivity from business intelligence (BI) tools such as Power BI Desktop and Microsoft Excel. The Timestream for LiveAnalytics ODBC driver is currently available on Windows, macOS and Linux, and also supports SSO with Okta and Microsoft Azure Active Directory (AD).

For more information, see Amazon Timestream for LiveAnalytics ODBC driver documentation on GitHub.

Topics

- Setting up the Timestream for LiveAnalytics ODBC driver
- Connection string syntax and options for the ODBC driver
- Connection string examples for the Timestream for LiveAnalytics ODBC driver
- Troubleshooting connection with the ODBC driver

Setting up the Timestream for LiveAnalytics ODBC driver

Set up access to Timestream for LiveAnalytics in your AWS account

If you haven't already set up your AWS account to work with Timestream for LiveAnalytics, follow the insructions in Accessing Timestream for LiveAnalytics.

Install the ODBC driver on your system

Download the appropriate Timestream ODBC driver installer for your system from the ODBC GitHub repository, and follow the installation instructions that apply to your system:.

- Windows installation guide
- MacOS installation guide
- Linux installation guide

Set up a data source name (DSN) for the ODBC driver

Follow the instructions in the DSN configuration guide for your system:

- Windows DSN configuration
- MacOS DSN configuration
- Linux DSN configuration

Set up your business intelligence (BI) application to work with the ODBC driver

Here are instructions for setting several common BI applications to work with the ODBC driver:

- Setting up Microsoft Power BI.
- Setting up Microsoft Excel
- Setting up Tableau

For other applications

Connection string syntax and options for the ODBC driver

The syntax for specifying connection-string options for the ODBC driver is as follows:

```
DRIVER={Amazon Timestream ODBC Driver};(option)=(value);
```

Available options are as follows:

Driver connection options

- **`Driver`** *(required)* – The driver being used with ODBC.

  The default is Amazon Timestream.
- **`DSN`** – The data source name (DSN) to use for configuring the connection.

  The default is `NONE`.
- **`Auth`** – The authentication mode. This must be one of the following:
  - `AWS_PROFILE` – Use the default credential chain.
  - `IAM` – Use AWS IAM credentials.
  - `AAD` – Use the Azure Active Directory (AD) identity provider.
  - `OKTA` – Use the Okta identity provider.

  The default is `AWS_PROFILE`.

Endpoint configuration options

- **`EndpointOverride`** – The endpoint override for the Timestream for LiveAnalytics service. This is an advanced option that overrides the region. For example:

  ```
  query-cell2.timestream.us-east-1.amazonaws.com
  ```
- **`Region`** – The signing region for the Timestream for LiveAnalytics service endpoint.

  The default is `us-east-1`.

Credentials provider option

- **`ProfileName`** – The profile name in the AWS config file.

  The default is `NONE`.

AWS IAM authentication options

- **`UID`** or **`AccessKeyId`** – The AWS user access key id. If both `UID` and `AccessKeyId` are provided in the connection string, the `UID` value will be used unless it is empty.

  The default is `NONE`.
- **`PWD`** or **`SecretKey`** – The AWS user secret access key. If both `PWD` and `SecretKey` are provided in the connection string, the `PWD` value with will be used unless it's empty.

  The default is `NONE`.
- **`SessionToken`** – The temporary session token required to access a database with multi-factor authentication (MFA) enabled. Do not include a trailing `=` in the input.

  The default is `NONE`.

SAML-based authentication options for Okta

- **`IdPHost`** – The hostname of the specified IdP.

  The default is `NONE`.
- **`UID`** or **`IdPUserName`** – The user name for the specified IdP account. If both `UID` and `IdPUserName` are provided in the connection string, the `UID` value will be used unless it's empty.

  The default is `NONE`.
- **`PWD`** or **`IdPPassword`** – The password for the specified IdP account. If both `PWD` and `IdPPassword` are provided in the connection string, the `PWD` value will be used unless it's empty.

  The default is `NONE`.
- **`OktaApplicationID`** – The unique Okta-provided ID associated with the Timestream for LiveAnalytics application. A place to find the application ID (AppId) is in the `entityID` field provided in the application metadata. An example is:

  ```
  entityID="http://www.okta.com//(IdPAppID)
  ```

  The default is `NONE`.
- **`RoleARN`** – The Amazon Resource Name (ARN) of the role that the caller is assuming.

  The default is `NONE`.

- **`IdPARN`** – The Amazon Resource Name (ARN) of the SAML provider in IAM that describes the IdP.

  The default is `NONE`.

SAML-based authentication options for Azure Active Directory

- **`UID`** or **`IdPUserName`** – The user name for the specified IdP account..

  The default is `NONE`.
- **`PWD`** or **`IdPPassword`** – The password for the specified IdP account.

  The default is `NONE`.
- **`AADApplicationID`** – The unique id of the registered application on Azure AD.

  The default is `NONE`.
- **`AADClientSecret`** – The client secret associated with the registered application on Azure AD used to authorize fetching tokens.

  The default is `NONE`.
- **`AADTenant`** – The Azure AD Tenant ID.

  The default is `NONE`.
- **`RoleARN`** – The Amazon Resource Name (ARN) of the role that the caller is assuming.

  The default is `NONE`.
- **`IdPARN`** – The Amazon Resource Name (ARN) of the SAML provider in IAM that describes the IdP.

  The default is `NONE`.

AWS SDK (advanced) Options

- **`RequestTimeout`** – The time in milliseconds that the AWS SDK waits for a query request before timing out. Any non-positive value disables the request timeout.

  The default is `3000`.

- **`ConnectionTimeout`** – The time in milliseconds that the AWS SDK waits for data to be transferred over an open connection before timing out. A value of 0 disables the connection timeout. This value must not be negative.

  The default is `1000`.
- **`MaxRetryCountClient`** – The maximum number of retry attempts for retryable errors with 5xx error codes in the SDK. The value must not be negative.

  The default is `0`.
- **`MaxConnections`** – The maximum number of allowed concurrently open HTTP connections to the Timestream service. The value must be positive.

  The default is 25.

ODBC driver logging Options

- **`LogLevel`** – The log level for driver logging. Must be one of:
  - **0** (OFF).
  - **1** (ERROR).
  - **2** (WARNING).
  - **3** (INFO).
  - **4** (DEBUG).

  The default is 1 (ERROR).

  Warning: personal information could be logged by the driver when using the DEBUG logging mode.
- **`LogOutput`** – Folder in which to store the log file.

  The default is:
  - **Windows:** `%USERPROFILE%`, or if not available, `%HOMEDRIVE%%HOMEPATH%`.
  - **macOS and Linux:** `$HOME`, or if not available, the field `pw_dir` from the function `getpwuid(getuid())` return value.

SDK logging options

The AWS SDK log level is separate from the Timestream for LiveAnalytics ODBC driver log level. Setting one does not affect the other.

The SDK Log Level is set using the environment variable `TS_AWS_LOG_LEVEL`. Valid values are:

- `OFF`
- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`
- `FATAL`

If `TS_AWS_LOG_LEVEL` is not set, the SDK log level is set to the default, which is `WARN`.

Connecting through a proxy

The ODBC driver supports connecting to Amazon Timestream for LiveAnalytics through a proxy. To use this feature, configure the following environment variables based on your proxy setting:

- **`TS_PROXY_HOST`** – the proxy host.
- **`TS_PROXY_PORT`** – The proxy port number.
- **`TS_PROXY_SCHEME`** – The proxy scheme, either `http` or `https`.
- **`TS_PROXY_USER`** – The user name for proxy authentication.
- **`TS_PROXY_PASSWORD`** – The user password for proxy authentication.
- **`TS_PROXY_SSL_CERT_PATH`** – The SSL Certificate file to use for connecting to an HTTPS proxy.
- **`TS_PROXY_SSL_CERT_TYPE`** – The type of the proxy client SSL certificate.
- **`TS_PROXY_SSL_KEY_PATH`** – The private key file to use for connecting to an HTTPS proxy.
- **`TS_PROXY_SSL_KEY_TYPE`** – The type of the private key file used to connect to an HTTPS proxy.
- **`TS_PROXY_SSL_KEY_PASSWORD`** – The passphrase to the private key file used to connect to an HTTPS proxy.

Connection string examples for the Timestream for LiveAnalytics ODBC driver

Example of connecting to the ODBC driver with IAM credentials

```
Driver={Amazon Timestream ODBC Driver};Auth=IAM;AccessKeyId=(your access key
 ID);secretKey=(your secret key);SessionToken=(your session token);Region=us-east-2;
```

Example of connecting to the ODBC driver with a profile

```
Driver={Amazon Timestream ODBC Driver};ProfileName=(the profile name);region=us-west-2;
```

The driver will attempt to connect using the credentials provided in `~/.aws/credentials`, or if a file is specified in the environment variable `AWS_SHARED_CREDENTIALS_FILE`, using the credentials in that file.

Example of connecting to the ODBC driver with Okta

```
driver={Amazon Timestream ODBC Driver};auth=okta;region=us-west-2;idPHost=(your host at
 Okta);idPUsername=(your user name);idPPassword=(your password);OktaApplicationID=(your
 Okta AppId);roleARN=(your role ARN);idPARN=(your Idp ARN);
```

Example of connecting to the ODBC driver with Azure Active Directory (AAD)

```
driver={Amazon Timestream ODBC Driver};auth=aad;region=us-west-2;idPUsername=(your
 user name);idPPassword=(your password);aadApplicationID=(your AAD
 AppId);aadClientSecret=(your AAD client secret);aadTenant=(your AAD
 tenant);roleARN=(your role ARN);idPARN=(your idP ARN);
```

Example of connecting to the ODBC driver with a specified endpoint and a log level of 2 (WARNING)

```
Driver={Amazon Timestream ODBC Driver};Auth=IAM;AccessKeyId=(your access
 key ID);secretKey=(your secret key);EndpointOverride=ingest.timestream.us-
west-2.amazonaws.com;Region=us-east-2;LogLevel=2;
```

Troubleshooting connection with the ODBC driver

> **Note**
>
> When the username and password are already specified in the DSN, there is no need to specify them again when the ODBC driver manager asks for them.

An error code of `01S02` with a message, `Re-writing` *`(connection string option)` `(have you specified it several times?`* occurs when a connection string option is passed more than once in the connection string. Specifying an option more than once raises an error. When making a connection with a DSN and a connection string, if a connection option is already specified in the DSN, do not specify it again in the connection string.

VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon Timestream for LiveAnalytics by creating an *interface VPC endpoint*. For more information, see VPC endpoints (AWS PrivateLink).

Best practices

To fully realize the benefits of the Amazon Timestream for LiveAnalytics, follow the best practices described below.

> **Note**
>
> When running proof-of-concept applications, consider the amount of data your application will accumulate over a few months or years while evaluating the performance and scale of Timestream for LiveAnalytics. As your data grows over time, you'll notice that the performance of Timestream for LiveAnalytics remains mostly unchanged because its serverless architecture can leverage massive amounts of parallelism for processing larger data volumes and automatically scale to match needs of your application.

Topics

- Data modeling

- Security
- Configuring Amazon Timestream for LiveAnalytics
- Writes
- Queries
- Scheduled queries
- Client applications and supported integrations
- General

Data modeling

Amazon Timestream for LiveAnalytics is designed to collect, store, and analyze time series data from applications and devices emitting a sequence of data with a timestamp. For optimal performance, the data being sent to Timestream for LiveAnalytics must have temporal characteristics and time must be a quintessential component of the data.

Timestream for LiveAnalytics provides you the flexibility to model your data in different ways to suit your application's requirements. In this section, we cover several of these patterns and provide guidelines for you to optimize your costs and performance. Familiarize yourself with key Amazon Timestream for LiveAnalytics concepts such as dimensions and measures. In this section, you will learn more about the following when deciding whether to create a single table or multiple tables to store data:

- Which data to put in the same table vs. when you want to separate data across multiple tables and databases.
- How to choose between Timestream for LiveAnalytics multi-measure records compared to single-measure records, and the benefits of modeling using multi-measure records especially when your application is tracking multiple measurements at the same time instant.
- Which attributes to model as dimensions or as measures.
- How to effectively use the measure name attributes to optimize your query latency.

Topics

- Single table vs. multiple tables
- Multi-measure records vs. single-measure records
- Dimensions and measures

- Using measure name with multi-measure records
- Recommendations for partitioning multi-measure records

Single table vs. multiple tables

As you are modeling your data in application, another important aspect is how to model the data into tables and databases. Databases and tables in Timestream for LiveAnalytics are abstractions for access control, specifying KMS keys, retention periods, and so on. Timestream for LiveAnalytics automatically partitions your data and is designed to scale resources to match the ingestion, storage, and query load and requirements for your applications.

A table in Timestream for LiveAnalytics can scale to petabytes of data stored and tens of gigabytes per second of data writes. Queries can process hundreds of terabytes per hour. Queries in Timestream for LiveAnalytics can span multiple tables and databases, providing joins and unions to provide seamless access to your data across multiple tables and databases. So scale of data or request volumes are usually not the primary concern when deciding how to organize your data in Timestream for LiveAnalytics. Below are some important considerations when deciding which data to co-locate in the same table compared to in different tables, or tables in different databases.

- Data retention policies (memory store retention, magnetic store retention, etc.) are supported at the granularity of a table. Therefore, data that requires different retention policies needs to be in different tables.
- AWS KMS keys that are used to encrypt your data are configured at the database level. Therefore, different encryption key requirements imply the data will need to be in different databases.
- Timestream for LiveAnalytics supports resource-based access control at the granularity of tables and databases. Consider your access control requirements when deciding which data you write to the same table vs. different tables.
- Be aware of the limits on the number of dimensions, measure names, and multi-measure attribute names when deciding which data is stored in which table.
- Consider your query workload and access patterns when deciding how you organize your data, as the query latency and ease of writing your queries will be dependent on that.
  - If you store data that you frequently query in the same table, that will generally ease the way you write your queries so that you can often avoid having to write joins, unions, or common table expressions. This also usually results in lower query latency. You can use predicates on dimensions and measure names to filter the data that is relevant to the queries.

For instance, consider a case where you store data from devices located in six continents. If your queries frequently access data from across continents to get a global aggregated view, then storing data from these continents in the same table will result in easier to write queries. On the other hand, if you store data on different tables, you still can combine the data in the same query, however, you will need to write a query to union the data from across tables.

- Timestream for LiveAnalytics uses adaptive partitioning and indexing on your data so queries only get charged for data that is relevant to your queries. For instance, if you have a table storing data from a million devices across six continents, if your query has predicates of the form `WHERE device_id = 'abcdef'` or `WHERE continent = 'North America'`, then queries are only charged for data for the device or for the continent.
- Wherever possible, if you use measure name to separate out data in the same table that is not emitted at the same time or not frequently queried, then using predicates such as `WHERE measure_name = 'cpu'` in your query, not only do you get the metering benefits, Timestream for LiveAnalytics can also effectively eliminate partitions that do not have the measure name used in your query predicate. This enables you to store related data with different measure names in the same table without impacting query latency or costs, and avoids spreading the data into multiple tables. The measure name is essentially used to partition the data and prune partitions irrelevant to the query.

Multi-measure records vs. single-measure records

Timestream for LiveAnalytics allows you to write data with multiple measures per record (multi-measure) or single measure per record (single-measure).

Multi-measure records

In many use cases, a device or an application you are tracking may emit multiple metrics or events at the same timestamp. In such cases, you can store all the metrics emitted at the same timestamp in the same multi-measure record. That is, all the measures stored in the same multi-measure record appear as different columns in the same row of data.

Consider, for instance, that your application is emitting metrics such as cpu, memory, and disk_iops from a device measured at the same time instant. The following is an example of such a table where multiple metrics emitted at the same time instant are stored in the same row. You will that see two hosts are emitting the metrics once every second.

Hostname	measure_name	Time	cpu	Memory	disk_iops
host-24Gju	metrics	2021-12-01 19:00:00	35	54.9	38.2
host-24Gju	metrics	2021-12-01 19:00:01	36	58	39
host-28Gju	metrics	2021-12-01 19:00:00	15	55	92
host-28Gju	metrics	2021-12-01 19:00:01	16	50	40

Single-measure records

The single-measure records are suitable when your devices emit different metrics at different time periods, or you are using custom processing logic that emits metrics/events at different time periods (for instance, when a device's reading/state changes). Because every measure has a unique timestamp, the measures can be stored in their own records in Timestream for LiveAnalytics. For instance, consider an IoT sensor, which tracks soil temperature and moisture, that emits a record only when it detects a change from the previous reported entry. The following example provides an example of such data being emitted using single measure records.

device_id	measure_name	Time	measure_value::double	measure_value::bigint
sensor-sea478	temperature	2021-12-01 19:22:32	35	NULL
sensor-sea478	temperature	2021-12-01 18:07:51	36	NULL
sensor-sea478	moisture	2021-12-01 19:05:30	NULL	21

device_id	measure_name	Time	measure_value::double	measure_value::bigint
sensor-sea478	moisture	2021-12-01 19:00:01	NULL	23

Comparing single-measure and multi-measure records

Timestream for LiveAnalytics provides you the flexibility to model your data as single-measure or multi-measure records depending on your application's requirements and characteristics. A single table can store both single-measure and multi-measure records if your application requirements so desire. In general, when your application is emitting multiple measures/events at the same time instant, then modeling the data as multi-measure records is usually recommended for performant data access and cost-effective data storage.

For instance, if you consider a DevOps use case tracking metrics and events from hundreds of thousands of servers, each server periodically emits 20 metrics and 5 events, where the events and metrics are emitted at the same time instant. That data can be modeled either using single-measure records or using multi-measure records (see the open-sourced data generator for the resulting schema). For this use case, modeling the data using multi-measure records compared to single-measure records results in:

- *Ingestion metering* - Multi-measure records results in about *40 percent lower ingestion* bytes written.
- *Ingestion batching* - Multi-measure records result in bigger batches of data being sent, which implies the clients need fewer threads and fewer CPU to process the ingestion.
- *Storage metering* - Multi-measure records result in about *8X lower storage*, resulting in significant storage savings for both memory and magnetic store.
- *Query latency* - Multi-measure records results in lower query latency for most query types when compared to single-measure records.
- *Query metered bytes* - For queries scanning less than 10 MB data, both single-measure and multi-measure records are comparable. For queries accessing a single measure and scanning > 10 MB data, single measure records usually results in lower bytes metered. For queries referencing three or more measures, multi-measure records result in lower bytes metered.

- *Ease of expressing multi-measure queries* - When your queries reference multiple measures, modeling your data with multi-measure records results in easier to write and more compact queries.

The previous factors will vary depending on how many metrics you are tracking, how many dimensions your data has, etc. While the preceding example provides some concrete data for one example, we see across many application scenarios and use cases where if your application emits multiple measures at the same instant, storing data as multi-measure records is more effective. Moreover, multi-measure records provide you the flexibility of data types and storing multiple other values as context (for example, storing request IDs, and additional timestamps, which is discussed later).

Note that a multi-measure record can also model sparse measures such as the previous example for single-measure records: you can use the `measure_name` to store the name of the measure and use a generic multi-measure attribute name, such as `value_double` to store `DOUBLE` measures, `value_bigint` to store `BIGINT` measures, `value_timestamp` to store additional `TIMESTAMP` values, and so on.

Dimensions and measures

A table in Timestream for LiveAnalytics allows you to store *dimensions* (identifying attributes of the device/data you are storing) and *measures* (the metrics/values you are tracking); see Amazon Timestream for LiveAnalytics concepts for more details. As you are modeling your application on Timestream for LiveAnalytics, how you map your data into dimensions and measures impacts your ingestion and query latency. The following are guidelines on how to model your data as dimensions and measures that you can apply to your use case.

Choosing dimensions

Data that identifies the source that is sending the time series data is a natural fit for dimensions, which are attributes that don't change over time. For instance, if you have a server emitting metrics, then the attributes identifying the server, such as hostname, Region, rack, and Availability Zone, are candidates for dimensions. Similarly, for an IoT device with multiple sensors reporting time series data, attributes such as device ID and sensor ID are candidates for dimensions.

If you are writing data as multi-measure records, dimensions and multi-measure attributes appear as columns in the table when you do a `DESCRIBE` or run a `SELECT` statement on the table. Therefore, when writing your queries, you can freely use the dimensions and measures in the same

query. However, as you construct your write record to ingest data, keep the following in mind as you choose which attributes are specified as dimensions and which ones are measure values:

- The dimension names, dimension values, measure name, and timestamp uniquely identify the time series data. Timestream for LiveAnalytics uses this unique identifier to automatically de-duplicate data. That is, if Timestream for LiveAnalytics receives two data points with the same values of dimension names, dimension values, measure name, and timestamp, and the values have the same version number, then Timestream for LiveAnalytics de-duplicates. If the new write request has a lower version than the data already existing in Timestream for LiveAnalytics, the write request is rejected. If the new write request has a higher version, then the new value overwrites the old value. Therefore, how you choose your dimension values will impact this de-duplication behavior.
- Dimension names and values cannot be updated, but measure value can be. Therefore, any data that might need updates is better modeled as measure values. For instance, if you have a machine on the factory floor whose color can change, you can model the color as a measure value, unless you also want to use the color as an identifying attribute that is needed for de-duplication. That is, measure values can be used to store attributes that only slowly change over time.

Note that a table in Timestream for LiveAnalytics does not limit the number of unique combinations of dimension names and values. For instance, you can have billions of such unique value combinations stored in a table. However, as you will see with the following examples, careful choice of dimensions and measures can significantly optimize your request latency, especially for queries.

Unique IDs in dimensions

If your application scenario requires you to store a unique identifier for every data point (for example, a request ID, a transaction ID, or a correlation ID), modeling the ID attribute as a measure value will result in significantly better query latency. When modeling your data with multi-measure records, the ID appears in the same row in context with your other dimensions and time series data, so your queries can continue to use them effectively. For instance, considering a DevOps use case where every data point emitted by a server has a unique request ID attribute, modeling the request ID as a measure value results in up to 4x lower query latency across different query types, as opposed to modeling the unique request ID as a dimension.

You can use the similar analogy for attributes that are not entirely unique for every data point, but have hundreds of thousands or millions of unique values. You can model those attributes both

as dimensions or measure values. You would want to model it as a dimension if the values are necessary for de-duplication on the write path as discussed earlier or you often use it as a predicate (for example, in the `WHERE` clause with an equality predicate on a value of that attribute such as `device_id = 'abcde'` where your application is tracking millions of devices) in your queries.

Richness of data types with multi-measure records

Multi-measure records provide you the flexibility to effectively model your data. Data that you store in a multi-measure record appear as columns in the table similar to dimensions, thus providing the same ease of querying for dimension and measure values. You saw some of these patterns in the examples discussed earlier. Below you will find additional patterns to effectively use multi-measure records to meet your application's use cases.

Multi-measure records support attributes of data types `DOUBLE`, `BIGINT`, `VARCHAR`, `BOOLEAN`, and `TIMESTAMP`. Therefore, they naturally fit different types of attributes:

- *Location information*: For instance, if you want to track a location (expressed as latitude and longitude), then modeling it as a multi-measure attribute will result in lower query latency compared to storing them as `VARCHAR` dimensions, especially when you have predicates on the latitudes and longitudes.
- *Multiple timestamps in a record*: If your application scenario requires you to track multiple timestamps for a time series record, you can model them as additional attributes in the multi-measure record. This pattern can be used to store data with future timestamps or past timestamps. Note that every record will still use the timestamp in the time column to partition, index, and uniquely identify a record.

In particular, if you have numeric data or timestamps on which you have predicates in the query, modeling those attributes as multi-measure attributes as opposed to dimensions will result in lower query latency. This is because when you model such data using the rich data types supported in multi-measure records, you can express the predicates using native data types instead of casting values from `VARCHAR` to another data type if you modeled such data as dimensions.

Using measure name with multi-measure records

Tables in Timestream for LiveAnalytics support a special attribute (or column) called *measure name*. You specify a value for this attribute for every record you write to Timestream for LiveAnalytics. For single-measure records, it is natural to use the name of your metric (such as CPU or memory for server metrics, or temperature or pressure for sensor metrics). When using multi-

measure records, attributes in a multi-measure record are named and these names become column names in the table. Therefore, cpu, memory, temperature, and pressure can become multi-measure attribute names. A natural question is how to effectively use the measure name.

Timestream for LiveAnalytics uses the values in the measure name attribute to partition and index the data. Therefore, if a table has multiple different measure names, and if the queries use those values as query predicates, then Timestream for LiveAnalytics can use its custom partitioning and indexing to prune out data that is not relevant to queries. For instance, if your table has `cpu` and `memory` measure names, and your query has a predicate `WHERE measure_name = 'cpu'`, Timestream for LiveAnalytics can effectively prune data for measure names not relevant to the query, for example, rows with measure name memory in this example. This pruning applies even when using measure names with multi-measure records. You can use the measure name attribute effectively as a partitioning attribute for a table. Measure name along with dimension names and values, and time are used to partition the data in a Timestream for LiveAnalytics table. Be aware of the limits on the number of unique measure names allowed in a Timestream for LiveAnalytics table. Also note that a measure name is associated with a measure value data type as well. For example, a single measure name can only be associated with one type of measure value. That type can be one of `DOUBLE`, `BIGINT`, `BOOLEAN`, `VARCHAR`, or `MULTI`. Multi-measure records stored with a measure name will have the data type of `MULTI`. Since a single multi-measure record can store multiple metrics with different data types (`DOUBLE`, `BIGINT`, `VARCHAR`, `BOOLEAN`, and `TIMESTAMP`), you can associate data of different types in a multi-measure record.

The following sections describe a few different examples of how the measure name attribute can be effectively used to group together different types of data in the same table.

IoT sensors reporting quality and value

Consider you have an application monitoring data from IoT sensors. Each sensor tracks different measures, such as temperature and pressure. In addition to the actual values, the sensors also report the quality of the measurements, which is a measure of how accurate the reading is, and a unit for the measurement. Since quality, unit, and value are emitted together, they can be modeled as multi-measure records, as shown in the example data below where `device_id` is a dimension, and `quality`, `value`, and `unit` are multi-measure attributes:

device_id	measure_name	Time	Quality	Value	Unit
sensor-sea478	temperature	2021-12-01 19:22:32	92	35	c
sensor-sea478	temperature	2021-12-01 18:07:51	93	34	c
sensor-sea478	pressure	2021-12-01 19:05:30	98	31	psi
sensor-sea478	pressure	2021-12-01 19:00:01	24	132	psi

This approach allows you to combine the benefits of multi-measure records along with partitioning and pruning data using the values of measure name. If queries reference a single measure, such as temperature, then you can include a `measure_name` predicate in the query. The following is an example of such a query, which also projects the unit for measurements whose quality is above 90.

```
SELECT device_id, time, value AS temperature, unit
FROM db.table
WHERE time > ago(1h)
    AND measure_name = 'temperature'
    AND quality > 90
```

Using the `measure_name` predicate on the query enables Timestream for LiveAnalytics to effectively prune partitions and data that is not relevant to the query, thus improving your query latency.

It is also possible to have all of the metrics stored in the same multi-measure record if all the metrics are emitted at the same timestamp and/or multiple metrics are queried together in the same query. For instance, you can construct a multi-measure record with attributes such as temperature_quality, temperature_value, temperature_unit, pressure_quality, pressure_value, and pressure_unit. Many of the points discussed earlier about modeling data using single-measure vs. multi-measure records apply in your decision of how to model the data. Consider your query access patterns and how your data is generated to choose a model that optimizes your cost, ingestion and query latency, and ease of writing your queries.

Different types of metrics in the same table

Another use case where you can combine multi-measure records with measure name values is to model different types of data that are independently emitted from the same device. Consider the DevOps monitoring use case where servers are emitting two types of data: regularly emitted metrics and irregular events. An example of this approach is the schema discussed in the data generator modeling a DevOps use case. In this case, you can store the different types of data emitted from the same server in the same table by using different measure names. For instance, all the metrics that are emitted at the same time instant are stored with measure name metrics. All the events that are emitted at a different time instant from the metrics are stored with measure name events. The measure schema for the table (for example, output of `SHOW MEASURES` query) is:

measure_name	data_type	Dimensions
events	multi	[{"data_type":"varchar","dimension_name":"availability_zone"},{"data_type":"varchar","dimension_name":"microservice_name"},{"data_type":"varchar","dimension_name":"instance_name"},{"data_type":"varchar","dimension_name":"process_name"},{"data_type":"varchar","dimension_name":"jdk_version"},{"data_type":"varchar","dimension_name":"cell"},{"data_type":"varchar","dimension_name":"region"},{"data_type":"varchar","dimension_name":"silo"}]
metrics	multi	[{"data_type":"varchar","dimension_name":"availability_zone"},{"data_type":"varchar","dimension_name":"microservice_name"},{"data_type"

measure_name	data_type	Dimensions
		:"varchar","dimension_name":"instance_name"},{"data_type":"varchar","dimension_name":"os_version"},{"data_type":"varchar","dimension_name":"cell"},{"data_type":"varchar","dimension_name":"region"},{"data_type":"varchar","dimension_name":"silo"},{"data_type":"varchar","dimension_name":"instance_type"}]

In this case, you can see that the events and metrics also have different sets of dimensions, where events have different dimensions `jdk_version` and `process_name` while metrics have dimensions `instance_type` and `os_version`.

Using different measure names allow you to write queries with predicates such as `WHERE measure_name = 'metrics'` to get only the metrics. Also having all the data emitted from the same instance in the same table implies you can also write a simpler query with the `instance_name` predicate to get all data for that instance. For instance, a predicate of the form `WHERE instance_name = 'instance-1234'` without a `measure_name` predicate will return all data for a specific server instance.

Recommendations for partitioning multi-measure records

> **Important**
>
> **This section is deprecated!**
> These recommendations are out of date. Partitioning is now better controlled using customer-defined partition keys.

We have seen that there is a growing number of workloads in the time series ecosystem that require ingesting and storing massive amounts of data while simultaneously needing low latency query responses when accessing data by a high cardinality set of dimension values.

Because of such characteristics, recommendations in this section will be useful for customer workloads that have the following:

- Adopted or want to adopt multi-measure records.
- Expect to have a high volume of data coming into the system that will be stored for long periods.
- Require low latency response times for their main access (query) patterns.
- Know that the most important queries patterns involve a filtering condition of some sort in the predicate. This filtering condition is based around a high cardinality dimension. For example, consider events or aggregations by UserId, DeviceId, ServerID, host-name, and so forth.

In these cases, a single name for all the multi-measure measures will not help since our engine uses multi-measure names to partition the data and having a single value limits the partition advantage that you get. The partitioning for these records is mainly based on two dimensions. Let's say time is on the x-axis and a hash of dimension names and the `measure_name` is on the y-axis. The `measure_name` in these cases works almost like a partitioning key.

Our recommendation is as follows:

- When modeling your data for use cases like the one we mentioned, use a `measure_name` that is a direct derivative of your main query access pattern. For example:
  - Your use case requires tracking application performance and QoE from the end user point of view. This could also be tracking measurements for a single server or IoT device.
  - If you are querying and filtering by UserId, then you need, at ingestion time, to find the best way to associate `measure_name` to UserId.
  - Since a multi-measure table can only hold 8,192 different measure names, whatever formula is adopted should not generate more that 8,192 different values.
- One approach that we have applied with success for string values is to apply a hashing algorithm to the string value. Then perform the modulo operation with the absolute value of the hash result and 8,192.

```
measure_name = getMeasureName(UserId)
int getMeasureName(value) {
    hash_value =  abs(hash(value))
```

```
    return hash_value % 8192
}
```

- We also added `abs()` to remove the sign eliminating the possibility for values to range from -8,192 to 8,192. This should be performed prior to the modulo operation.
- By using this method your queries can run on a fraction of the time that would take to run on an unpartitioned data model.
- When querying the data, make sure that you include a filtering condition in the predicate that uses the newly derived value of the `measure_name`. For example:
  - ```
    SELECT * FROM your_database.your_table
    WHERE host_name = 'Host-1235' time BETWEEN '2022-09-01'
        AND '2022-09-18'
        AND measure_name = (SELECT
     cast(abs(from_big_endian_64(xxhash64(CAST('HOST-1235' AS varbinary))))%8192 AS
     varchar))
    ```
  - This will minimize the total number of partitions scanned to get you data that will translate in faster queries over time.

Keep in mind that if you want to obtain the benefits from this partition schema, the hash needs to be calculated on the client side and passed to Timestream for LiveAnalytics as a static value to the query engine. The preceding example provides a way to validate that the generated hash can be resolved by the engine when needed.

time	host_name	location	server_type	cpu_usage	available_memory	cpu_temp
2022-09-07 21:48:44 .000	host-1235	us-east1	5.8xl	55	16.2	78
R2022-09-07 21:48:44 .000	host-3587	us-west1	5.8xl	62	18.1	81
    ```

time	host_name	location	server_type	cpu_usage	available_memory	cpu_temp
2022-09-07 21:48:45.000000000	host-258743	eu-central	5.8xl	88	9.4	91
2022-09-07 21:48:45 .000	host-35654	us-east2	5.8xl	29	24	54
R2022-09-07 21:48:45 .000	host-254	us-west1	5.8xl	44	32	48

To generate the associated `measure_name` following our recommendation, there are two paths that depend on your ingestion pattern.

1. *For batch ingestion of historical data*—You can add the transformation to your write code if you will use your own code for the batch process.

   Building on top of the preceding example.

```
List<String> hosts = new ArrayList<>();

hosts.add("host-1235");
hosts.add("host-3587");
hosts.add("host-258743");
hosts.add("host-35654");
hosts.add("host-254");

for (String h: hosts){
    ByteBuffer buf2 = ByteBuffer.wrap(h.getBytes());
    partition = abs(hasher.hash(buf2, 0L)) % 8192;
    System.out.println(h + " - " + partition);

}
```

Output

```
host-1235 - 6445
host-3587 - 6399
host-258743 - 640
host-35654 - 2093
host-254 - 7051
```

Resulting dataset

time	host_name	location	measure_name	server_type	cpu_usage	available_memory	cpu_temp
2022-09-07 21:48:44 .0	host-1235	us-east1	6445	5.8xl	55	16.2	78
R2022-09-07 21:48:44 .0	host-3587	us-west1	6399	5.8xl	62	18.1	81
2022-09-07 21:48:45. 00000000	host-258743	eu-central	640	5.8xl	88	9.4	91
2022-09-07 21:48:45 .0	host-35654	us-east2	2093	5.8xl	29	24	54
R2022-09-07 21:48:45 .0	host-254	us-west1	7051	5.8xl	44	32	48

2. *For real-time ingestion*—You need to generate the `measure_name` in-flight as data is coming in.

In both cases, we recommend you test your hash generating algorithm at both ends (ingestion and querying) to make sure you are getting the same results.

Here are some code examples to generate the hashed value based on `host_name`.

Example Python

```
>>> import xxhash
>>> from bitstring import BitArray
>>> b=xxhash.xxh64('HOST-ID-1235').digest()
>>> BitArray(b).int % 8192
### 3195
```

Example Go

```
package main

import (
    "bytes"
    "fmt"
    "github.com/cespare/xxhash"
)

func main() {
    buf := bytes.NewBufferString("HOST-ID-1235")
    x := xxhash.New()
    x.Write(buf.Bytes())
    // convert unsigned integer to signed integer before taking mod
    fmt.Printf("%f\n", abs(int64(x.Sum64())) % 8192)
}

func abs(x int64) int64 {
    if (x < 0) {
        return -x
    }
    return x
}
```

Example Java

```
import java.nio.ByteBuffer;

import net.jpountz.xxhash.XXHash64;

public class test {
    public static void main(String[] args) {
        XXHash64 hasher = net.jpountz.xxhash.XXHashFactory.fastestInstance().hash64();

        String host = "HOST-ID-1235";
        ByteBuffer buf = ByteBuffer.wrap(host.getBytes());

        Long result = Math.abs(hasher.hash(buf, 0L));
        Long partition = result % 8192;

        System.out.println(result);
        System.out.println(partition);
    }
}
```

Example dependency in Maven

```
        <dependency>
            <groupId>net.jpountz.lz4</groupId>
            <artifactId>lz4</artifactId>
            <version>1.3.0</version>
        </dependency>
```

Security

- For continuous access to Timestream for LiveAnalytics, ensure that encryption keys are secured and are not revoked or made inaccessible.
- Monitor API access logs from AWS CloudTrail. Audit and revoke any anomalous access pattern from unauthorized users.
- Follow additional guidelines described in Security best practices for Amazon Timestream for LiveAnalytics.

Configuring Amazon Timestream for LiveAnalytics

Configure the data retention period for the memory store and the magnetic store to match the data processing, storage, query performance, and cost requirements.

- Set the data retention of the memory store to match your application's requirements for processing late-arriving data. Late-arriving data is incoming data with a timestamp earlier than the current time. It is emitted from resources that batch events for a time period before sending the data to Timestream for LiveAnalytics, and also from resources with intermittent connectivity e.g. an IoT sensor that is online intermittently.
- If you expect late-arriving data to occasionally arrive with timestamps earlier than the memory store retention, you should enable magnetic store writes for your table. Once you set the EnableMagneticStoreWrites in MagneticStoreWritesProperties for a table, the table will accept data with timestamp earlier than your memory store retention but within your magnetic store retention period.
- Consider the characteristics of queries that you plan to run on Timestream for LiveAnalytics such as the types of queries, frequency, time range, and performance requirements. This is because the memory store and magnetic store are optimized for different scenarios. The memory store is optimized for fast point-in-time queries that process small amounts of recent data sent to Timestream for LiveAnalytics. The magnetic store is optimized for fast analytical queries that process medium to large volumes of data sent to Timestream for LiveAnalytics.
- Your data retention period should also be influenced by the cost requirements of your system.

  For example, consider a scenario where the late-arriving data threshold for your application is 2 hours and your applications send many queries that process a day's-worth, week's-worth, or month's-worth of data. In that case, you may want to configure a smaller retention period for the memory store (2-3 hours) and allow more data to flow to the magnetic store given the magnetic store is optimized for fast analytical queries.

Understand the impact of increasing or decreasing the data retention period of the memory store and the magnetic store of an existing table.

- When you decrease the retention period of the memory store, the data is moved from the memory store to the magnetic store, and this data transfer is permanent. Timestream for LiveAnalytics does not retrieve data from the magnetic store to populate the memory store. When you decrease the retention period of the magnetic store, the data is deleted from the system, and the data deletion is permanent.

- When you increase the retention period of the memory store or the magnetic store, the change takes effect for data being sent to Timestream for LiveAnalytics from that point onwards. Timestream for LiveAnalytics does not retrieve data from the magnetic store to populate the memory store. For example, if the retention period of the memory store was initially set to 2 hours and then increased to 24 hours, it will take 22 hours for the memory store to contain 24 hours worth of data.

Writes

- Ensure that the timestamp of the incoming data is not earlier than data retention configured for the memory store and no later than the future ingestion period defined in Quotas. Sending data with a timestamp outside these bounds will result in the data being rejected by Timestream for LiveAnalytics unless you enable magnetic store writes for your table. If you enable magnetic store writes, ensure that the timestamp for incoming data is not earlier than data retention configured for the magnetic store.
- If you expect late arriving data, turn on magnetic store writes for your table. This will allow ingestion for data with timestamps that fall outside your memory store retention period but still within your magnetic store retention period. You can set this by updating the `EnableMagneticStoreWrites` flag in the `MagneticStoreWritesProperties` for your table. This property is false by default. Note that writes to the magnetic store will not be immediately available to query. They will be available within 6 hours.
- Target high throughput workloads to the memory store by ensuring the timestamps of the ingested data fall within the memory store retention bounds. Writes to the magnetic store are limited to a max number of active magnetic store partitions that can receive concurrent ingestion for a database. You can see this `ActiveMagneticStorePartitions` metric in CloudWatch. To reduce active magnetic store partitions, aim to reduce the number of series and duration of time you ingest into concurrently for magnetic store ingestion.
- While sending data to Timestream for LiveAnalytics, batch multiple records in a single request to optimize data ingestion performance.
  - It is beneficial to batch together records from the same time series and records with the same measure name.
  - Batch as many records as possible in a single request as long as the requests are within the service limits defined in Quotas.
  - Use common attributes where possible to reduce data transfer and ingestion costs. For more information, see WriteRecords API.

- If you encounter partial client-side failures while writing data to Timestream for LiveAnalytics, you can resend the batch of records that failed ingestion after you've addressed the rejection cause.
- Data ordered by timestamps has better write performance.
- Amazon Timestream for LiveAnalytics is designed to automatically scale to the needs of your application. When Timestream for LiveAnalytics notices spikes in write requests from your application, your application may experience some level of initial memory store throttling. If your application experiences memory store throttling, continue sending data to Timestream for LiveAnalytics at the same (or increased) rate to enable Timestream for LiveAnalytics to automatically scale to satisfy the needs of your application. If you see magnetic store throttling, you should decrease your rate of magnetic store ingestion until your number of `ActiveMagneticStorePartitions` falls.

Batch load

Best practices for batch load are described in Batch load best practices.

Queries

Following are suggested best practices for queries with Amazon Timestream for LiveAnalytics.

- Include only the measure and dimension names essential to query. Adding extraneous columns will increase data scans, which impacts the performance of queries.
- Before deploying your query in production, we recommend that you review query insights to make sure that the spatial and temporal pruning is optimal. For more information, see Using query insights to optimize queries in Amazon Timestream.
- Where possible, push the data computation to Timestream for LiveAnalytics using the built-in aggregates and scalar functions in the SELECT clause and WHERE clause as applicable to improve query performance and reduce cost. See SELECT and Aggregate functions.
- Where possible, use approximate functions. E.g., use APPROX_DISTINCT instead of COUNT(DISTINCT column_name) to optimize query performance and reduce the query cost. See Aggregate functions.
- Use a CASE expression to perform complex aggregations instead of selecting from the same table multiple times. See The CASE statement.

- Where possible, include a time range in the WHERE clause of your query. This optimizes query performance and costs. For example, if you only need the last one hour of data in your dataset, then include a time predicate such as time > ago(1h). See SELECT and Interval and duration.
- When a query accesses a subset of measures in a table, always include the measure names in the WHERE clause of the query.
- Where possible, use the equality operator when comparing dimensions and measures in the WHERE clause of a query. An equality predicate on dimensions and measure names allows for improved query performance and reduced query costs.
- Wherever possible, avoid using functions in the WHERE clause to optimize for cost.
- Refrain from using LIKE clause multiple times. Rather, use regular expressions when you are filtering for multiple values on a string column. See Regular expression functions.
- Only use the necessary columns in the GROUP BY clause of a query.
- If the query result needs to be in a specific order, explicitly specify that order in the ORDER BY clause of the outermost query. If your query result does not require ordering, avoid using an ORDER BY clause to improve query performance.
- Use a LIMIT clause if you only need the first N rows in your query.
- If you are using an ORDER BY clause to look at the top or bottom N values, use a LIMIT clause to reduce the query costs.
- Use the pagination token from the returned response to retrieve the query results. For more information, see Query.
- If you've started running a query and realize that the query will not return the results you're looking for, cancel the query to save cost. For more information, see CancelQuery.
- If your application experiences throttling, continue sending data to Amazon Timestream for LiveAnalytics at the same rate to enable Amazon Timestream for LiveAnalytics to auto-scale to the satisfy the query throughput needs of your application.
- If the query concurrency requirements of your applications exceed the default limits of Timestream for LiveAnalytics, contact Support for limit increases.

Scheduled queries

Scheduled queries help you optimize your dashboards by pre-computing some fleet-wide aggregate statistics. So a natural question to ask is how do you take your use case and identify which results to pre-compute and how to use these results stored in a derived table to create your

dashboard. The first step in this process is to identify which panels to pre-compute. Below are some high-level guidelines:

- Consider the bytes scanned by the queries that are used to populate the panels, the frequency of dashboard reload, and number of concurrent users who would load these dashboards. You should start with the dashboards loaded most frequently and scanning significant amounts of data. The first two dashboards in the aggregate dashboard example as well as the aggregate dashboard in the drill down example are good examples of such dashboards.
- Consider which computations are being repeatedly used. While it is possible to create a scheduled query for every panel and every variable value used in the panel, you can significantly optimize your costs and the number of scheduled queries by looking for avenues to use one computation to pre-compute the data necessary for multiple panels.
- Consider the frequency of your scheduled queries to refresh the materialized results in the derived table. You would want to analyze how frequently a dashboard is refreshed vs. the time window that is queried in a dashboard vs. the time binning used in the pre-computation as well as the panels in the dashboards. For instance, if a dashboard that is plotting hourly aggregates for the past few days is only refreshed once in a few hours, you might want to configure your scheduled queries to only refresh once every 30 mins or an hour. On the other hand, if you have a dashboard that plots per minute aggregates and is refreshed every minute or so, you would want your scheduled queries to refresh the results every minute or few minutes.
- Consider which query patterns can be further optimized (both from a query cost and query latency perspective) using scheduled queries. For instance, when computing the unique dimension values frequently used as variables in dashboards, or returning the last data point emitted from a sensor or the first data point emitted from a sensor after a certain date, etc. Some of these example patterns are discussed in this guide.

The preceding considerations will have a significant impact on your savings when you move your dashboard to query the derived tables, the freshness of data in your dashboards, and the cost incurred by the scheduled queries.

Client applications and supported integrations

Run your client application from the same Region as Timestream for LiveAnalytics to reduce network latencies and data transfer costs. For more information about working with other services, see Working with other services. The following are some other helpful links.

- Best Practices for AWS Development with the AWS SDK for Java

- Best practices for working with AWS Lambda functions
- Best Practices for Amazon Managed Service for Apache Flink
- Best practices for creating dashboards in Grafana

General

- Ensure that you follow the The AWS Well-Architected Framework when using Timestream for LiveAnalytics. This whitepaper provides guidance around best practices in operational excellence, security, reliability, performance efficiency, and cost optimization.

Metering and cost optimization

With Amazon Timestream for LiveAnalytics, you pay only for what you use. Timestream for LiveAnalytics meters separately for writes, data stored, and data scanned by queries. The price of each metering dimension is specified on the pricing page. You can estimate your monthly bill using the Amazon Timestream for LiveAnalytics Pricing Calculator.

This section describes how metering works for writes, storage and queries in Timestream for LiveAnalytics. Example scenarios and calculations are also provided. In addition, a list of best practices for cost optimization is included. You can select a topic below:

Topics

- Writes
- Storage
- Queries
- Cost optimization
- Monitoring with Amazon CloudWatch

Writes

The write size of each time series event is calculated as the sum of the size of the timestamp and one or more dimension names, dimension values, measure names, and measure values. The size of the timestamp is 8 bytes. The size of dimension names, dimension values, and measure names are the length of the UTF-8 encoded bytes of the string representing each dimension name, dimension value, and measure name. The size of the measure value depends on the data type. It is 1 byte for

the boolean data type, 8 bytes for bigint and double, and the length of the UTF-8 encoded bytes for strings. Each write is counted in units of 1 KiB.

Two example calculations are provided below:

Topics

- Calculating the write size of a time series event
- Calculating the number of writes

Calculating the write size of a time series event

Consider a time series event representing the CPU utilization of an EC2 instance as shown below:

Time	region	az	vpc	Hostname	measure_name	measure_value::double
1602983435238563000	us-east-1	1d	vpc-1a2b3c4d	host-24Gju	cpu_utilization	35.0

The write size of the time series event can be calculated as:

- time = 8 bytes
- first dimension = 15 bytes (`region+us-east-1`)
- second dimension = 4 bytes (`az+1d`)
- third dimension = 15 bytes (`vpc+vpc-1a2b3c4d`)
- fourth dimension = 18 bytes (`hostname+host-24Gju`)
- name of the measure = 15 bytes (`cpu_utilization`)
- value of the measure = 8 bytes

Write size of the time series event = 83 bytes

Calculating the number of writes

Now consider 100 EC2 instances, similar to the instance described in Calculating the write size of a time series event, emitting metrics every 5 seconds. The total monthly writes for the EC2 instances will vary based on how many time series events exist per write and if common attributes are being used while batching time series events. An example of calculating total monthly writes is provided for each of the following scenarios:

Topics

- One time series event per write
- Batching time series events in a write
- Batching time series events and using common attributes in a write

One time series event per write

If each write contains only one time series event, the total monthly writes are calculated as:

- 100 time series events = 100 writes every 5 seconds
- x 12 writes/minute = 1,200 writes
- x 60 minutes/hour = 72,000 writes
- x 24 hours/day = 1,728,000 writes
- x 30 days/month = 51,840,000 writes

Total monthly writes = 51,840,000

Batching time series events in a write

Given each write is measured in units of 1 KB, a write can contain a batch of 12 time series events (998 bytes) and the total monthly writes are calculated as:

- 100 time series events = 9 writes (12 time series events per write) every 5 seconds
- x 12 writes/minute = 108 writes
- x 60 minutes/hour = 6,480 writes
- x 24 hours/day = 155,520 writes
- x 30 days/month = 4,665,600 writes

Total monthly writes = 4,665,600

Batching time series events and using common attributes in a write

If the region, az, vpc, and measure name are common across 100 EC2 instances, the common values can be specified just once per write and are referred to as common attributes. In this case, the size of common attributes is 52 bytes, and the size of the time series events is 27 bytes. Given each write is measured in units of 1 KiB, a write can contain 36 time series events and common attributes, and the total monthly writes are calculated as:

- 100 time series events = 3 writes (36 time series events per write) every 5 seconds
- x 12 writes/minute = 36 writes
- x 60 minutes/hour = 2,160 writes
- x 24 hours/day = 51,840 writes
- x 30 days/month = 1,555,200 writes

Total monthly writes = 1,555,200

> **Note**
>
> Due to usage of batching, common attributes and rounding of the writes to units of 1KB, the storage size of the time series events may be different than write size.

Storage

The storage size of each time series event in the memory store and the magnetic store is calculated as the sum of the size of the timestamp, dimension names, dimension values, measure names, and measure values. The size of the timestamp is 8 bytes. The size of dimension names, dimension values, and measure names are the length of the UTF-8 encoded bytes of each string representing the dimension name, dimension value, and measure name. The size of the measure value depends on the data type. It is 1 byte for boolean data types, 8 bytes for bigint and double, and the length of the UTF-8 encoded bytes for strings. Each measure is stored as a separate record in Amazon Timestream for LiveAnalytics, i.e. if your time series event has four measures, there will be four records for that time series event in storage.

Considering the example of the time series event representing the CPU utilization of an EC2 instance (see Calculating the write size of a time series event), the storage size of the time series event is calculated as:

- time = 8 bytes
- first dimension = 15 bytes (`region+us-east-1`)
- second dimension = 4 bytes (`az+1d`)
- third dimension = 15 bytes (`vpc+vpc-1a2b3c4d`)
- fourth dimension = 18 bytes (`hostname+host-24Gju`)
- name of the measure = 15 bytes (`cpu_utilization`)
- value of the measure = 8 bytes

Storage size of the time series event = 83 bytes

> **Note**
>
> The memory store is metered in GB-hour and the magnetic store is metered in GB-month.

Queries

Queries are charged based on the duration of Timestream compute units (TCUs) used by your application in TCU-hours as specified on the Amazon Timestream pricing page. Amazon Timestream for LiveAnalytics' query engine prunes irrelevant data while processing a query. Queries with projections and predicates including time ranges, measure names, and/or dimension names enable the query processing engine to prune a significant amount of data and help with lowering query costs.

Cost optimization

To optimize the cost of writes, storage, and queries, use the following best practices with Amazon Timestream for LiveAnalytics:

- Batch multiple time series events per write to reduce the number of write requests.
- Consider using Multi-measure records, which allows you to write multiple time-series measures in a single write request and stores your data in a more compact manner. This reduces the number of write requests as well as data storage cost and query cost.

- Use common attributes with batching to batch more time series events per write to further reduce the number of write requests.
- Set the data retention of the memory store to match your application's requirements for processing late-arriving data. Late-arriving data is incoming data with a timestamp earlier than the current time and outside the memory store retention period.
- Set the data retention of the magnetic store to match your long term data storage requirements.
- While writing queries, include only the measure and dimension names essential to query. Adding extraneous columns will increase data scans and therefore will also increase the query cost. We recommend that you review query insights to assess the pruning efficiency of the included dimensions and measures.
- Where possible, include a time range in the WHERE clause of your query. For example, if you only need the last one hour of data in your dataset, include a time predicate such as `time > ago(1h)`.
- When a query accesses a subset of measures in a table, always include the measure names in the WHERE clause of the query.
- If you've started running a query and realize that the query will not return the results you're looking for, cancel the query to save on cost.

Monitoring with Amazon CloudWatch

You can monitor Timestream for LiveAnalytics using Amazon CloudWatch, which collects and processes raw data from Timestream for LiveAnalytics into readable, near-real-time metrics. It records these statistics for two weeks so that you can access historical information and gain a better perspective on how your web application or service is performing. By default, Timestream for LiveAnalytics metric data is automatically sent to CloudWatch in 1-minute or 15-minute periods. For more information, see What Is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*.

Topics

- How do I use Timestream for LiveAnalytics metrics?
- Timestream for LiveAnalytics metrics and dimensions
- Creating CloudWatch alarms to monitor Timestream for LiveAnalytics

How do I use Timestream for LiveAnalytics metrics?

The metrics reported by Timestream for LiveAnalytics provide information that you can analyze in different ways. The following list shows some common uses for the metrics. These are suggestions to get you started, not a comprehensive list.

How can I?	Relevant metrics
`How can I determine if any system errors occurred?`	You can monitor `SystemErrors` to determine whether any requests resulted in a server error code. Typically, this metric should be equal to zero. If it isn't, you might want to investiga te.
`How can I monitor the amount of data in the memory store?`	You can monitor `MemoryCumulativeBytesMetered` over the specified time period, to monitor the amount of data stored in memory store in bytes. This metric is emitted every hour and you can track the bytes stored at an account as well as at database granularity. The memory store is metered in GB-hour (the cost of storing 1GB of data for one hour). So multiplying the hourly value of `MemoryCumulativeBy tesMetered` with GB-hour pricing in your Region will give you the cost incurred per hour. Dimensions: Operation (storage), DatabaseName, Metric name
`How can I monitor the amount of data in the magnetic store?`	You can monitor `MagneticCumulativeBytesMetered` over the specified time period, to monitor the amount of data stored in magnetic store in bytes. This metric is emitted every hour and you can track the bytes stored at an account as well as at database granularity. The memory store is metered in GB-month (the cost of storing 1GB of data for one month). So multiplying the hourly value of `MagneticCumulative BytesMetered` with GB-month pricing in your Region will give you the cost incurred per hour. For example, if the value of `MagneticCumulativeBytesMetered` is 107374182 400 bytes (100GB), then the hourly charge of 1GB of data in magnetic store = (0.03) (us-east-1 pricing) / (30.4*24). Multiplyi

How can I?	Relevant metrics
	ng this value with the `MagneticCumulativeBytesMetered` in GB will give ~$0.004 for that hour. Dimensions: Operation (Storage), DatabaseName, Metric name
`How can I monitor the data scanned by queries?`	You can monitor `CumulativeBytesMetered` over the specified time period, to monitor the data scanned by queries (in bytes) sent to Timestream for LiveAnalytics. This metric is emitted after the query execution and you can track the data scanned at account and database granularity. You can calculate the query cost for a particular period by multiplying the value of the metric with per GB scanned pricing in your Region. The bytes scanned by scheduled queries are accounted for in this metric. Dimensions: Operation (Query), DatabaseName, Metric name
`How can I monitor the data scanned by scheduled queries?`	You can monitor `CumulativeBytesMetered` over the specified time period, to monitor the data scanned by scheduled queries (in bytes) executed by Timestream for LiveAnalytics. This metric is emitted after the query execution and you can track the data scanned at account and database granularity. You can calculate the query cost for a particula r period by multiplying the value of the metric with per GB scanned pricing in your Region. **Note** The bytes metered are also accounted for in the query `CumulativeBytesMetered` . Dimensions: Operation (TriggeredScheduledQuery), DatabaseN ame, Metric name

How can I?	Relevant metrics
How can I monitor the number of records ingested?	You can monitor `NumberOfRecords` over the specified time period to monitor the number of records ingested. You can track the bytes stored at an account as well as at database granularity. You can also use this metric to monitor the writes made by Scheduled Queries when query results are written into a separate table. When using the `WriteRecords` API, the metric is emitted for each `WriteRecords` request, with the CloudWatch Operation dimension being `WriteRecords`. When using the `BatchLoad` or `ScheduledQuery` APIs, the metric is emitted at intervals determined by the service until the task completes. The CloudWatch Operation dimension for this metric is either `BatchLoad` or `ScheduledQuery`, depending on which API is used. Dimensions: Operation (WriteRecords, BatchLoad, or Scheduled Query), DatabaseName, Metric name

How can I?	Relevant metrics
How can I monitor the cost of records ingested?	You can monitor CumulativeBytesMetered to monitor the number of bytes ingested that accrue cost. You can track the bytes stored at an account as well as at database granularity. Ingested records are metered in cumulative bytes. Multiplying the value of CumulativeBytesMetered by Writes pricing in your Region gives you the ingestion cost incurred. When using the WriteRecords API, this metric is emitted for each WriteRecords request, with the CloudWatch Operation dimension being WriteRecords . When using the BatchLoad or ScheduledQuery API, the metric is emitted at intervals determined by the service until the task completes. The CloudWatch Operation dimension for this metric is BatchLoad or ScheduledQuery depending on which API is used.. Dimensions: Operation (WriteRecords, BatchLoad, or Scheduled Query), DatabaseName, Metric name
How can I monitor the Timestream Compute Units (TCUs) used in my account?	You can monitor QueryTCU over the desired time period, to monitor the compute units provisioned in your account. This metric is emitted every 15-minutes. Units: Count Valid Statistics: Minimum, Maximum Metric: ResourceCount Dimensions: Service: Timestream , Namespace:AWS/Usage , Resource: QueryTCU, Type: Resource, Class: OnDemand

How can I?	Relevant metrics
How can I monitor the number of provisioned Timestream Compute Units (TCUs) used in my account?	**Note** Provisioned TCU is available only in the Asia Pacific (Mumbai) region. You can monitor `QueryTCU` to monitor the number of provisioned TCUs used for query workload in the account. This metric is emitted every minute for the during active query workload from the account. Units: `Count` Valid Statistics: Minimum, Maximum Metric: `ResourceCount` Dimensions: `Service: Timestream` , `Namespace: AWS/Usage` , `Resource: ProvisionedQueryTCU` , `Class: None`

How can I?	Relevant metrics
How can I monitor the provisioned Timestream Compute Units (TCUs) used in my account?	**Note** Provisioned TCU is available only in the Asia Pacific (Mumbai) region. You can monitor `QueryTCU` over the specified time period, to monitor the compute units consumed for query workload in the account. This metric is emitted with maximum and minimum compute units for every minute during active query workload from the account. Units: `Count` Valid Statistics: Minimum, Maximum Metric: `ResourceCount` Dimensions: `Service: Timestream` , `Namespace: AWS/Usage` , `Resource: QueryTCU`, `Class: Provisioned`

Timestream for LiveAnalytics metrics and dimensions

When you interact with Timestream for LiveAnalytics, it sends the following metrics and dimensions to Amazon CloudWatch. All metrics are aggregated and reported every minute. You can use the following procedures to view the metrics for Timestream for LiveAnalytics.

To view metrics using the CloudWatch console

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. If necessary, change the Region. On the navigation bar, choose the Region where your AWS resources reside. For more information, see AWS Service Endpoints.
3. In the navigation pane, choose **Metrics**.

4. Under the **All metrics** tab, choose `AWS/Timestream for LiveAnalytics.`

To view metrics using the AWS CLI

- At a command prompt, use the following command.

```
aws cloudwatch list-metrics --namespace "AWS/Timestream"
```

Dimensions for Timestream for LiveAnalytics metrics

The metrics for Timestream for LiveAnalytics are qualified by the values for the account, table name, or operation. You can use the CloudWatch console to retrieve Timestream for LiveAnalytics data along any of the dimensions in the following table:

Dimension	Description
`DatabaseName`	This dimension limits the data to a specific Timestream for LiveAnalytics database. This value can be any database in the current Region and the current AWS account
`Operation`	This dimension limits the data to one of the Timestream for LiveAnalytics operations, such as `Storage`, `WriteRecords` , `BatchLoad` , or `ScheduledQuery` . See the Timestream for LiveAnalytics Query API Reference for a list of available values.
`TableName`	This dimension limits the data to a specific table in a Timestrea m for LiveAnalyticss database.

> **Important**
>
> `CumulativeBytesMetered`, `UserErrors` and `SystemErrors` metrics only have the `Operation` dimension. `SuccessfulRequestLatency` metrics always have `Operation` dimension, but may also have the `DatabaseName` and `TableName` dimensions too, depending on the value of `Operation`. This is because Timestream for LiveAnalytics table-

level operations have `DatabaseName` and `TableName` as dimensions, but account level operations do not.

Timestream for LiveAnalytics metrics

> **Note**
>
> Amazon CloudWatch aggregates all the following Timestream for LiveAnalytics metrics at one-minute intervals.

General metrics

Metric	Description
`SuccessfulRequestLatency`	The successful requests to Timestream for LiveAnalytics during the specified time period. SuccessfulRequestLatency can provide two different kinds of information: • The elapsed time for successful requests (Minimum, Maximum,Sum, or Average). • The number of successful requests (SampleCount). SuccessfulRequestLatency reflects activity only within Timestream for LiveAnalytics and does not take into account network latency or client-side activity. Units: `Milliseconds` **Dimensions** • `DatabaseName` • `TableName` • `Operation`

Metric	Description
	Valid Statistics: • `Minimum` • `Maximum` • `Average` • `SampleCount` • `P10` • `p50` • `p90` • `p95` • `p99`

Writing and storage metrics

Metric	Description
`MagneticStoreRejectedRecordCount`	The number of magnetic store written records that were rejected asynchronously. This can happen if the new record has a version that is less than the current version or the new record has version equal to the current version but has different data. Units: `Count` **Dimensions** • `DatabaseName` • `TableName` • `Operation` Valid Statistics: • `Sum`

Metric	Description
	• `SampleCount`
`MagneticStoreRejectedUpload UserFailures`	The number of magnetic store rejected record reports that were not uploaded due to user errors. This can be due to IAM permissions not configured correctly or a deleted S3 bucket. Units: `Count` **Dimensions** • `DatabaseName` • `TableName` • `Operation` Valid Statistics: • `Sum` • `SampleCount`
`MagneticStoreRejectedUpload SystemFailures`	The number of magnetic store rejected record reports that were not uploaded due to system errors. Units: `Count` **Dimensions** • `DatabaseName` • `TableName` • `Operation` Valid Statistics: • `Sum` • `SampleCount`

Metric	Description
`ActiveMagneticStorePartitions`	The number of magnetic store partitions actively ingesting data at a given time. Units: `Count` **Dimensions** • `DatabaseName` • `Operation` Valid Statistics: • `Sum` • `SampleCount`

Metric	Description
`MagneticStorePendingRecordsLatency`	The oldest write to a magnetic store that is not available for query. Records written to the magnetic store will be available for querying within 6 hours. Units: `Milliseconds` **Dimensions** • `DatabaseName` • `TableName` • `Operation` Valid Statistics: • `Minimum` • `Maximum` • `Average` • `SampleCount` • `P10` • `p50` • `p90` • `p95` • `p99`
`MemoryCumulativeBytesMetered`	The amount of data stored in memory store, in bytes Units: `Bytes` Dimensions: `Operation` Valid Statistics: • `Average`

Metric	Description
`MagneticCumulativeBytesMetered`	The amount of data stored in magnetic store, in bytes Units: `Bytes` Dimensions: `Operation` Valid Statistics: • `Average`
`CumulativeBytesMetered`	The amount of data metered by ingestion to Timestream for LiveAnalytics, in bytes. Units: `Bytes` Dimensions: `Operation` Valid Statistics: `Sum`
`NumberOfRecords`	The number of records ingested into Timestream for LiveAnalytics. Units: `Count` Dimensions: `Operation` Valid Statistics: `Sum`

Query metrics

Metric	Description
`CumulativeBytesMetered`	The amount of data scanned by queries sent to Timestream for LiveAnalytics, in bytes. Units: `Bytes` Dimensions: `Operation`

Metric	Description
	Valid Statistics: • `Sum`
`ResourceCount`	The Timestream Compute Units (TCUs) consumed for query workload in the account. This metric is emitted with maximum and minimum compute units for every minute during active query workload from the account. Units: `Count` Valid Statistics: Minimum, Maximum Dimensions: `Service: Timestream` , `Resource: QueryTCU`, `Type: Resource`, `Class: OnDemand`

Error metrics

Metric	Description
`SystemErrors`	The requests to Timestream for LiveAnalytics that generate a SystemError during the specified time period. A SystemError usually indicates an internal service error. Units: `Count` Dimensions: `Operation` Valid Statistics: • `Sum` • `SampleCount`

Metric	Description
`UserErrors`	Requests to Timestream for LiveAnalytics that generate an InvalidRequest error during the specified time period. An InvalidRequest usually indicates a client-side error, such as an invalid combination of parameters, an attempt to update a nonexistent table, or an incorrect request signature. UserErrors represents the aggregate of invalid requests for the current AWS Region and the current AWS account. Units: `Count` Dimensions: `Operation` Valid Statistics: • `Sum` • `SampleCount`

> **Important**
>
> Not all statistics, such as `Average` or `Sum`, are applicable for every metric. However, all of these values are available through the Timestream for LiveAnalytics console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics.

Creating CloudWatch alarms to monitor Timestream for LiveAnalytics

You can create an Amazon CloudWatch alarm for Timestream for LiveAnalytics that sends an Amazon Simple Notification Service (Amazon SNS) message when the alarm changes state. An alarm watches a single metric over a time period that you specify. It performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon SNS topic or Auto Scaling policy.

Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state. The state must have changed and been maintained for a specified number of periods.

For more information about creating CloudWatch alarms, see Using Amazon CloudWatch Alarms in the *Amazon CloudWatch User Guide*.

Troubleshooting

This section contains information on troubleshooting Timestream for LiveAnalytics.

Topics

- Handling WriteRecords throttles
- Handling rejected records
- Troubleshooting UNLOAD from Timestream for LiveAnalytics
- Timestream for LiveAnalytics specific error codes

Handling WriteRecords throttles

Your memory store write requests to Timestream may be throttled as Timestream scales to adapt to the data ingestion needs of your application. If your applications encounter throttling exceptions, you must continue to send data at the same (or higher) throughput to allow Timestream to automatically scale to your application's needs.

Your magnetic store write requests to Timestream may be throttled if the maximum limit of magnetic store partitions receiving ingestion. You will see a throttle message directing you to check the `ActiveMagneticStorePartitions` Cloudwatch metric for this database. This throttle may take up to 6 hours to resolve. To avoid this throttle, you should use the memory store for any high throughput ingestion workload. For magnetic store ingestion, you can target ingesting into fewer partitions by limiting how many series and the time duration that you ingest into

For more information about data ingestion best practices, see Writes.

Handling rejected records

If Timestream rejects records, you will receive a `RejectedRecordsException` with details about the rejection. Please refer to Handling write failure for more information on how to extract this information from the WriteRecords response.

All rejections will be included in this response **with the exception of updates to the magnetic store where the new record's version is less than or equal to the existing record's version**. In this case, Timestream will not update the existing record that has the higher version. Timestream will reject the new record with lower or equal version and write these errors asynchronously to your S3 bucket. In order to receive these asynchronous error reports, you should set the `MagneticStoreRejectedDataLocation` property in `MagneticStoreWriteProperties` on your table.

Troubleshooting UNLOAD from Timestream for LiveAnalytics

Following is guidance for troubleshooting related to the UNLOAD command.

Category	Error message	How to troubleshoot
S3 Key length	UNLOAD result file key when using the S3 prefix [%s] provided in the destination will exceed the S3 allowed key length. See documenta tion for more details.	When exporting query results using the UNLOAD statement, the S3 key length, comprising of sum of the length of S3 bucket name and prefix exceeds the maximum supported S3 key length. We recommend to reduce your prefix or bucket name length.
	UNLOAD result file key when using partitioned_by [%s] will exceed the S3 allowed key length. See documentation for more details.	When exporting query results using the UNLOAD statement, the S3 Key length using the partitioned_by column exceeds the maximum supported S3 key length. We recommend to partition with an alternate column or reduce the length of the partition ed_column (if feasible).
	UNLOAD result file key when using the S3 prefix [%s] along with the partitioned_by [%s]	When exporting query results using the UNLOAD statement , the S3 Key length, comprisin

Category	Error message	How to troubleshoot
	will exceed the S3 allowed key length. See documenta tion for more details.	g of sum of the length of S3 bucket name, the prefix, and the partitioned_by column name exceeds the maximum supported S3 key length. We recommend to reduce your prefix, bucket name length, or use an alternate column to partition your data.
	The generated S3 object key: %s is too long. See documentation for more details.	While processing your query using the `UNLOAD` statement , one of the values in the partitioned column exceeds the maximum supported S3 key length. The partition column and value can be found in the object key generated.

Category	Error message	How to troubleshoot
S3 throttles	We have detected that Amazon S3 is throttling the writes from UNLOAD command. See Amazon Timestream documentation for more information	Refer to S3 documentation here. S3 API call rate could be throttled when multiple readers/writers access the same folder. Please audit the call volume to the bucket provided. If you are using same bucket for multiple concurrent UNLOAD queries, try using different buckets for the same. If you are using same bucket for multiple operations other than Timestream for LiveAnalytics UNLOAD, consider moving UNLOAD results to separate bucket.

Timestream for LiveAnalytics specific error codes

This section contains the specific error codes for Timestream for LiveAnalytics.

Timestream for LiveAnalytics write API errors

InternalServerException

HTTP Status Code: 500

ThrottlingException

HTTP Status Code: 429

ValidationException

HTTP Status Code: 400

ConflictException

HTTP Status Code: 409

AccessDeniedException

You do not have sufficient access to perform this action.

HTTP Status Code: 403

ServiceQuotaExceededException

HTTP Status Code: 402

ResourceNotFoundException

HTTP Status Code: 404

RejectedRecordsException

HTTP Status Code: 419

InvalidEndpointException

HTTP Status Code: 421

Timestream for LiveAnalytics query API errors

ValidationException

HTTP Status Code: 400

QueryExecutionException

HTTP Status Code: 400

ConflictException

HTTP Status Code: 409

ThrottlingException

HTTP Status Code: 429

InternalServerException

HTTP Status Code: 500

InvalidEndpointException

HTTP Status Code: 421

Quotas

This topic describes current quotas, also referred to as limits, within Amazon Timestream for LiveAnalytics. Each quota applies on a per-Region basis unless otherwise specified.

Topics

- Default quotas
- Service limits
- Supported data types
- Batch load
- Naming constraints
- Reserved keywords
- System identifiers
- UNLOAD

Default quotas

The following table contains the Timestream for LiveAnalytics quotas and the default values.

displayName	Description	defaultValue
Databases per account	The maximum number of databases you can create per AWS account.	500
Tables per account	The maximum number of tables you can create per AWS account.	50000
Request rate for CRUD APIs	The maximum number of Create/Update/Delete requests allowed per second per account, in the current Region.	1

displayName	Description	defaultValue
Request rate for other APIs	The maximum number of List/Describe/Prepare/ExecuteScheduledQueryAPI requests allowed per second per account, in the current Region.	5
Scheduled queries per account	The maximum number of scheduled queries you can create per AWS account.	10000
Maximum count of active magnetic store partitions	The maximum number of active magnetic store partitions per database. A partition might remain active for up to six hours after receiving ingestion.	250

Service limits

The following table contains the Timestream for LiveAnalytics service limits and the default values. To edit data retention for a table from the console, see Edit a table.

displayName	Description	defaultValue
Future ingestion period in minutes	The maximum lead time (in minutes) for your time series data compared to the current system time. For example, if the future ingestion period is 15 minutes, then Timestream for LiveAnalytics will accept data that is up to 15 minutes	15

displayName	Description	defaultValue
	ahead of the current system time.	
Minimum retention period for memory store in hours	The minimum duration (in hours) for which data must be retained in the memory store per table.	1
Maximum retention period for memory store in hours	The maximum duration (in hours) for which data can be retained in the memory store per table.	8766
Minimum retention period for magnetic store in days	The minimum duration (in days) for which data must be retained in the magnetic store per table.	1
Maximum retention period for magnetic store in days	The maximum duration (in days) for which data can be retained in the magnetic store. This value is equivalent to 200 years.	73000
Default retention period for magnetic store in days	The default value (in days) for which data is retained in the magnetic store per table. This value is equivalent to 200 years.	73000
Default retention period for memory store in hours	The default duration (in hours) for which data is retained in the memory store.	6
Dimensions per table	The maximum number of dimensions per table.	128

displayName	Description	defaultValue
Measure names per table	The maximum number of unique measure names per table.	8192
Dimension name dimension value pair size per series	The maximum size of dimension name and dimension value pair per series.	2 Kilobytes
Maximum record size	The maximum size of a record.	2 Kilobytes
Records per WriteRecords API request	The maximum number of records in a WriteRecords API request.	100
Dimension name length	The maximum number of bytes for a Dimension name.	60 bytes
Measure name length	The maximum number of bytes for a Measure name.	256 bytes
Database name length	The maximum number of bytes for a Database name.	256 bytes
Table name length	The maximum number of bytes for a Table name.	256 bytes
QueryString length in KiB	The maximum length (in KiB) of a query string in UTF-8 encoded characters for a query.	256
Execution duration for queries in hours	The maximum execution duration (in hours) for a query. Queries that take longer will timeout.	1

displayName	Description	defaultValue
Query Insights	The maximum number of Query API requests allowed with query insights enabled per second per account, in the current Region.	1
Metadata size for query result	The maximum metadata size for a query result.	100 Kilobytes
Data size for query result	The maximum data size for a query result.	5 Gigabytes
Measures per multi-measure record	The maximum number of measures per multi-measure record.	256
Measure value size per multi-measure record	The maximum size of measure values per multi-measure record.	2048
Unique measures across multi-measure records per table	The unique measures in all the multi-measure records defined in a single table.	1024
Timestream Compute Units (TCUs) per account	The default maximum TCUs per account.	200

displayName	Description	defaultValue
Maximum Provisioned Timestream Compute Units (TCUs) per account. **Note** Provisioned TCU is available only in the Asia Pacific (Mumbai) region.	The maximum number of TCUs you can provision in your account.	1000
maxQueryTCU	The maximum query TCUs you can set for your account.	1000

Supported data types

The following table describes the supported data types for measure and dimension values.

Description	Timestream for LiveAnalytics value
Supported data types for measure values.	Big int, double, string, boolean, MULTI, Timestamp
Supported data types for dimension values.	String

Batch load

The current quotas, also referred to as limits, within batch load are as follows.

Description	Timestream for LiveAnalytics value
Max batch load task size	Max batch load task size cannot exceed 100 GB.

Description	Timestream for LiveAnalytics value
Files quantity	A batch load task cannot have more than 100 files.
Maximum file size	Maximum file size in a batch load task cannot exceed 5 GB.
CSV file row size	A row in a CSV file cannot exceed 16 MB. This is a hard limit which cannot be increased.
Active batch load tasks	A table cannot have more than 5 active batch load tasks and an account cannot have more than 10 active batch load tasks. Timestream for LiveAnalytics will throttle new batch load tasks until more resources are available.

Naming constraints

The following table describes naming constraints.

Description	Timestream for LiveAnalytics value
The maximum length of a dimension name.	60 bytes
The maximum length of a measure name.	256 bytes
The maximum length of a table name or database name.	256 bytes
Table and Database Name	• We recommend you do not use System identifiers. • Can contain a-z A-Z 0-9 _ (underscore) - (dash) . (dot). • All names must be encoded as UTF-8, and are case sensitive.

Description	Timestream for LiveAnalytics value
	Note Table and database names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.
Measure Name	• Must not contain System identifiers or colon ':'. • Must not start with a reserved prefix (`ts_`, `measure_value`). **Note** Table and database names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.
Dimension Name	• Must not contain System identifiers, colon ':' or double quote ("). • Must not start with a reserved prefix (`ts_`, `measure_value`). • Must not contain Unicode characters [0,31] listed here or "\u2028" or "\u2029". **Note** Dimension and measure names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.
All Column Names	Column names can not be duplicated. Since multi-measure records represent dimensions and measures as columns, the name for a dimension can not be the same as the name for a measure. Names are case sensitive.

Reserved keywords

All of the following are reserved keywords:

- ALTER
- AND
- AS
- BETWEEN
- BY
- CASE
- CAST
- CONSTRAINT
- CREATE
- CROSS
- CUBE
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- DEALLOCATE
- DELETE
- DESCRIBE
- DISTINCT
- DROP
- ELSE
- END
- ESCAPE
- EXCEPT
- EXECUTE
- EXISTS

- EXTRACT
- FALSE
- FOR
- FROM
- FULL
- GROUP
- GROUPING
- HAVING
- IN
- INNER
- INSERT
- INTERSECT
- INTO
- IS
- JOIN
- LEFT
- LIKE
- LOCALTIME
- LOCALTIMESTAMP
- NATURAL
- NORMALIZE
- NOT
- NULL
- ON
- OR
- ORDER
- OUTER
- PREPARE

- RECURSIVE
- RIGHT
- ROLLUP
- SELECT
- TABLE
- THEN
- TRUE
- UESCAPE
- UNION
- UNNEST
- USING
- VALUES
- WHEN
- WHERE
- WITH

System identifiers

We reserve column names "measure_value", "ts_non_existent_col" and "time" to be Timestream for LiveAnalytics system identifiers. Additionally, column names may not start with "ts_" or "measure_name". System identifiers are case sensitive. Identifiers compared using UTF-8 binary representation. This means that comparison for identifiers is case sensitive.

> **Note**
>
> System identifiers may not be used for dimension or measure names. We recommend you do not use system identifiers for database or table names.

UNLOAD

For limits related to the `UNLOAD` command, see Using UNLOAD to export query results to S3 from Timestream.

Query language reference

> **Note**
>
> This query language reference includes the following third-party documentation from the Trino Software Foundation (formerly Presto Software Foundation), which is licensed under the Apache License, Version 2.0. You may not use this file except in compliance with this license. To get a copy of the Apache License, Version 2.0, see the Apache website.

Timestream for LiveAnalytics supports a rich query language for working with your data. You can see the available data types, operators, functions and constructs below.

You can also get started right away with Timestream's query language in the Sample queries section.

Topics

- Supported data types
- Built-in time series functionality
- SQL support
- Logical operators
- Comparison operators
- Comparison functions
- Conditional expressions
- Conversion functions
- Mathematical operators
- Mathematical functions
- String operators
- String functions
- Array operators
- Array functions
- Bitwise functions
- Regular expression functions
- Date / time operators

- Date / time functions
- Aggregate functions
- Window functions
- Sample queries

Supported data types

Timestream for LiveAnalytics's query language supports the following data types.

> **Note**
>
> Data types supported for writes are described in Data types.

Data type	Description
`int`	Represents a 32-bit integer.
`bigint`	Represents a 64-bit signed integer.
`boolean`	One of the two truth values of logic, `True` and `False`.
`double`	Represents a 64-bit variable-precision data type. Implements IEEE Standard 754 for Binary Floating-Point Arithmetic. **Note** The query language is for reading data. There are functions for `Infinity` and `NaN` double values which can be used in queries. But you cannot write those values to Timestream.
`varchar`	Variable length character data with a maximum size of 2KB.
`array[`*T*`,...]`	Contains one or more elements of a specified data type *T*, where *T* can be any of the data types supported in Timestrea m.

Data type	Description
`row(T,...)`	Contains one or more named fields of data type *T*. The fields may be of any data type supported by Timestream, and are accessed with the dot field reference operator: `.`
`date`	Represents a date in the form *YYYY*-*MM*-*DD*. where *YYYY* is the year, *MM* is the month, and *DD* is the day, respectively. The supported range is from `1970-01-01` to `2262-04-11` . *Example:* `1971-02-03`
`time`	Represents the time of day in UTC. The `time` datatype is represented in the form *HH*.*MM*.*SS*.*sssssssss* . Supports nanosecond precision. *Example:* `17:02:07.496000000`
`timestamp`	Represents an instance in time using nanosecond precision time in UTC. *YYYY*-*MM*-*DD hh*:*mm*:*ss*.*sssssssss* Query supports timestamps in the range `1677-09-21 00:12:44.000000000` to `2262-04-11 23:47:16.854775807` .

Data type	Description

`interval`	Represents an interval of time as a string literal *Xt*, composed of two parts, *X* and *t*.
	
	X is an numeric value greater than or equal to `0`, and *t* is a unit of time like second or hour. The unit is not pluralized. The unit of time *t* is must be one of the following string literals:
	
	• `nanosecond`
	• `microsecond`
	• `millisecond`
	• `second`
	• `minute`
	• `hour`
	• `day`
	• `ns` (same as `nanosecond`)
	• `us` (same as `microsecond`)
	• `ms` (same as `millisecond`)
	• `s` (same as `second`)
	• `m` (same as `minute`)
	• `h` (same as `hour`)
	• `d` (same as `day`)
	
	Examples:
	
	`17s`
	
	`12second`
	
	`21hour`

Data type	Description
	2d
`timeseries[row(timestamp, T,...)]`	Represents the values of a measure recorded over a time interval as an `array` composed of `row` objects. Each `row` contains a `timestamp` and one or more measure values of data type *T*, where *T* can be any one of `bigint`, `boolean`, `double`, or `varchar`. Rows are assorted in ascending order by `timestamp`. The *timeseries* datatype represents the values of a measure over time.
unknown	Represents null data.

Built-in time series functionality

Timestream for LiveAnalytics provides built-in time series functionality that treat time series data as a first class concept.

Built-in time series functionality can be divided into two categories: views and functions.

You can read about each construct below.

Topics

- Timeseries views
- Time series functions

Timeseries views

Timestream for LiveAnalytics supports the following functions for transforming your data to the `timeseries` data type:

Topics

- CREATE_TIME_SERIES
- UNNEST

CREATE_TIME_SERIES

CREATE_TIME_SERIES is an aggregation function that takes all the raw measurements of a time series (time and measure values) and returns a timeseries data type. The syntax of this function is as follows:

```
CREATE_TIME_SERIES(time, measure_value::<data_type>)
```

where `<data_type>` is the data type of the measure value and can be one of bigint, boolean, double, or varchar. The second parameter cannot be null.

Consider the CPU utilization of EC2 instances stored in a table named **metrics** as shown below:

Time	region	az	vpc	instance_id	measure_name	measure_value::double
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	35.0
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	38.2
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	45.3
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	cpu_utilization	54.1
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	cpu_utilization	42.5

Time	region	az	vpc	instance_id	measure_name	measure_value::double
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	cpu_utilization	33.7

Running the query:

```
SELECT region, az, vpc, instance_id, CREATE_TIME_SERIES(time, measure_value::double) as
 cpu_utilization FROM metrics
    WHERE measure_name='cpu_utilization'
    GROUP BY region, az, vpc, instance_id
```

will return all series that have `cpu_utilization` as a measure value. In this case, we have two series:

region	az	vpc	instance_id	cpu_utilization
us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	[{time: 2019-12-04 19:00:00.000000000, measure_value::double: 35.0}, {time: 2019-12-04 19:00:01.000000000, measure_value::double: 38.2}, {time: 2019-12-04 19:00:02.000000000, measure_v

region	az	vpc	instance_id	cpu_utilization
				alue::double: 45.3}]
us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	[{time: 2019-12-04 19:00:00.000000000, measure_value::double: 35.1}, {time: 2019-12-04 19:00:01.000000000, measure_value::double: 38.5}, {time: 2019-12-04 19:00:02.000000000, measure_value::double: 45.7}]

UNNEST

UNNEST is a table function that enables you to transform `timeseries` data into the flat model. The syntax is as follows:

UNNEST transforms a `timeseries` into two columns, namely, `time` and `value`. You can also use aliases with UNNEST as shown below:

```
UNNEST(timeseries) AS <alias_name> (time_alias, value_alias)
```

where `<alias_name>` is the alias for the flat table, `time_alias` is the alias for the `time` column and `value_alias` is the alias for the `value` column.

For example, consider the scenario where some of the EC2 instances in your fleet are configured to emit metrics at a 5 second interval, others emit metrics at a 15 second interval, and you need the average metrics for all instances at a 10 second granularity for the past 6 hours. To get this data, you transform your metrics to the time series model using **CREATE_TIME_SERIES**. You can then use **INTERPOLATE_LINEAR** to get the missing values at 10 second granularity. Next, you transform the data back to the flat model using **UNNEST**, and then use **AVG** to get the average metrics across all instances.

```
WITH interpolated_timeseries AS (
    SELECT region, az, vpc, instance_id,
        INTERPOLATE_LINEAR(
            CREATE_TIME_SERIES(time, measure_value::double),
                SEQUENCE(ago(6h), now(), 10s)) AS interpolated_cpu_utilization
    FROM timestreamdb.metrics
    WHERE measure_name= ‘cpu_utilization’ AND time >= ago(6h)
    GROUP BY region, az, vpc, instance_id
)
SELECT region, az, vpc, instance_id, avg(t.cpu_util)
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_cpu_utilization) AS t (time, cpu_util)
GROUP BY region, az, vpc, instance_id
```

The query above demonstrates the use of **UNNEST** with an alias. Below is an example of the same query without using an alias for **UNNEST**:

```
WITH interpolated_timeseries AS (
    SELECT region, az, vpc, instance_id,
        INTERPOLATE_LINEAR(
            CREATE_TIME_SERIES(time, measure_value::double),
                SEQUENCE(ago(6h), now(), 10s)) AS interpolated_cpu_utilization
    FROM timestreamdb.metrics
    WHERE measure_name= ‘cpu_utilization’ AND time >= ago(6h)
    GROUP BY region, az, vpc, instance_id
)
SELECT region, az, vpc, instance_id, avg(value)
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_cpu_utilization)
GROUP BY region, az, vpc, instance_id
```

Time series functions

Amazon Timestream for LiveAnalytics supports timeseries functions, such as derivatives, integrals, and correlations, as well as others, to derive deeper insights from your time series data. This section provides usage information for each of these functions, as well as sample queries. Select a topic below to learn more.

Topics

- Interpolation functions
- Derivatives functions
- Integral functions
- Correlation functions
- Filter and reduce functions

Interpolation functions

If your time series data is missing values for events at certain points in time, you can estimate the values of those missing events using interpolation. Amazon Timestream supports four variants of interpolation: linear interpolation, cubic spline interpolation, last observation carried forward (locf) interpolation, and constant interpolation. This section provides usage information for the Timestream for LiveAnalytics interpolation functions, as well as sample queries.

Usage information

Function	Output data type	Description
`interpolate_linear (timeseries, array[timestamp])`	timeseries	Fills in missing data using linear interpolation.
`interpolate_linear (timeseries, timestamp)`	double	Fills in missing data using linear interpolation.

Function	Output data type	Description
`interpolate_spline_cubic(timeseries, array[timestamp])`	timeseries	Fills in missing data using cubic spline interpolation.
`interpolate_spline_cubic(timeseries, timestamp)`	double	Fills in missing data using cubic spline interpolation.
`interpolate_locf(timeseries, array[timestamp])`	timeseries	Fills in missing data using the last sampled value.
`interpolate_locf(timeseries, timestamp)`	double	Fills in missing data using the last sampled value.
`interpolate_fill(timeseries, array[timestamp], double)`	timeseries	Fills in missing data using a constant value.
`interpolate_fill(timeseries, timestamp, double)`	double	Fills in missing data using a constant value.

Query examples

Example

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using linear interpolation:

```
WITH binned_timeseries AS (
SELECT hostname, BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double),
 2) AS avg_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
```

```
    AND time > ago(2h)
GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
SELECT hostname,
    INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
            SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
FROM binned_timeseries
GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Example

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using interpolation based on the last observation carried forward:

```
WITH binned_timeseries AS (
SELECT hostname, BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double),
 2) AS avg_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
    AND time > ago(2h)
GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
SELECT hostname,
    INTERPOLATE_LOCF(
        CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
            SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
FROM binned_timeseries
GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Derivatives functions

Derivatives are used calculate the rate of change for a given metric and can be used to proactively respond to an event. For example, suppose you calculate the derivative of the CPU utilization of EC2 instances over the past 5 minutes, and you notice a significant positive derivative. This can be indicative of increased demand on your workload, so you may decide want to spin up more EC2 instances to better handle your workload.

Amazon Timestream supports two variants of derivative functions. This section provides usage information for the Timestream for LiveAnalytics derivative functions, as well as sample queries.

Usage information

Function	Output data type	Description
`derivative_linear(timeseries, interval)`	timeseries	Calculates the derivative of each point in the `timeseries` for the specified `interval`.
`non_negative_derivative_linear(timeseries, interval)`	timeseries	Same as `derivative_linear(timeseries, interval)`, but only returns positive values.

Query examples

Example

Find the rate of change in the CPU utilization every 5 minutes over the past 1 hour:

```
SELECT DERIVATIVE_LINEAR(CREATE_TIME_SERIES(time, measure_value::double), 5m) AS
 result
FROM “sampleDB”.DevOps
WHERE measure_name = 'cpu_utilization'
AND hostname = 'host-Hovjv' and time > ago(1h)
GROUP BY hostname, measure_name
```

Example

Calculate the rate of increase in errors generated by one or more microservices:

```
WITH binned_view as (
    SELECT bin(time, 5m) as binned_timestamp, ROUND(AVG(measure_value::double), 2) as
 value
    FROM “sampleDB”.DevOps
    WHERE micro_service = 'jwt'
    AND time > ago(1h)
    AND measure_name = 'service_error'
    GROUP BY bin(time, 5m)
)
SELECT non_negative_derivative_linear(CREATE_TIME_SERIES(binned_timestamp, value), 1m)
 as rateOfErrorIncrease
FROM binned_view
```

Integral functions

You can use integrals to find the area under the curve per unit of time for your time series events. As an example, suppose you're tracking the volume of requests received by your application per unit of time. In this scenario, you can use the integral function to determine the total volume of requests served per specified interval over a specific time period.

Amazon Timestream supports one variant of integral functions. This section provides usage information for the Timestream for LiveAnalytics integral function, as well as sample queries.

Usage information

Function	Output data type	Description

`integral_trapezoidal(timeseries(double))`
 	
`integral_trapezoidal(timeseries(double), interval day to second)`
	double	Approximates the integral per the specified `interval day to second` for the `timeseries` provided, using the trapezoidal rule. The interval day to second parameter is optional and the default is `1s`. For more

Function	Output data type	Description

`integral_trapezoidal(timeseries(bigint))`
 	
`integral_trapezoidal(timeseries(bigint), interval day to second)`
 	
`integral_trapezoidal(timeseries(integer), interval day to second)`
 	
`integral_trapezoidal(timeseries(integer))`
		information about intervals, see Interval and duration.

Query examples

Example

Calculate the total volume of requests served per five minutes over the past hour by a specific host:

```
SELECT INTEGRAL_TRAPEZOIDAL(CREATE_TIME_SERIES(time, measure_value::double), 5m) AS
 result FROM sample.DevOps
WHERE measure_name = 'request'
AND hostname = 'host-Hovjv'
AND time > ago (1h)
GROUP BY hostname, measure_name
```

Correlation functions

Given two similar length time series, correlation functions provide a correlation coefficient, which explains how the two time series trend over time. The correlation coefficient ranges from `-1.0` to `1.0`. `-1.0` indicates that the two time series trend in opposite directions at the same rate. whereas `1.0` indicates that the two timeseries trend in the same direction at the same rate. A value of `0`

indicates no correlation between the two time series. For example, if the price of oil increases, and the stock price of an oil company increases, the trend of the price increase of oil and the price increase of the oil company will have a positive correlation coefficient. A high positive correlation coefficient would indicate that the two prices trend at a similar rate. Similarly, the correlation coefficient between bond prices and bond yields is negative, indicating that these two values trends in the opposite direction over time.

Amazon Timestream supports two variants of correlation functions. This section provides usage information for the Timestream for LiveAnalytics correlation functions, as well as sample queries.

Usage information

Function	Output data type	Description
`correlate_pearson(timeseries, timeseries)`	double	Calculates Pearson's correlation coefficient for the two `timeseries` . The timeseries must have the same timestamps.
`correlate_spearman(timeseries, timeseries)`	double	Calculates Spearman's correlation coefficient for the two `timeseries` . The timeseries must have the same timestamps.

Query examples

Example

```
WITH cte_1 AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, measure_value::double),
        SEQUENCE(min(time), max(time), 10m)) AS result
    FROM sample.DevOps
    WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv' AND time > ago(1h)
    GROUP BY hostname, measure_name
```

```
),
cte_2 AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, measure_value::double),
        SEQUENCE(min(time), max(time), 10m)) AS result
    FROM sample.DevOps
    WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv' AND time > ago(1h)
    GROUP BY hostname, measure_name
)
SELECT correlate_pearson(cte_1.result, cte_2.result) AS result
FROM cte_1, cte_2
```

Filter and reduce functions

Amazon Timestream supports functions for performing filter and reduce operations on time series data. This section provides usage information for the Timestream for LiveAnalytics filter and reduce functions, as well as sample queries.

Usage information

Function	Output data type	Description
`filter(timeseries(T), function(T, Boolean))`	timeseries(T)	Constructs a time series from an the input time series, using values for which the passed `function` returns `true`.
`reduce(timeseries(T), initialState S, inputFunction(S, T, S), outputFunction(S, R))`	R	Returns a single value, reduced from the time series. The `inputFunction` will be invoked on each element in timeseries in order. In addition to taking the current element, inputFunction takes the current state (initiall y `initialState`) and returns the new state. The `outputFunction` will be

Function	Output data type	Description
		invoked to turn the final state into the result value. The `outputFunction` can be an identity function.

Query examples

Example

Construct a time series of CPU utilization of a host and filter points with measurement greater than 70:

```
WITH time_series_view AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, ROUND(measure_value::double,2)),
            SEQUENCE(ago(15m), ago(1m), 10s)) AS cpu_user
    FROM sample.DevOps
    WHERE hostname = 'host-Hovjv' and measure_name = 'cpu_utilization'
        AND time > ago(30m)
    GROUP BY hostname
)
SELECT FILTER(cpu_user, x -> x.value > 70.0) AS cpu_above_threshold
from time_series_view
```

Example

Construct a time series of CPU utilization of a host and determine the sum squared of the measurements:

```
WITH time_series_view AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, ROUND(measure_value::double,2)),
            SEQUENCE(ago(15m), ago(1m), 10s)) AS cpu_user
    FROM sample.DevOps
    WHERE hostname = 'host-Hovjv' and measure_name = 'cpu_utilization'
        AND time > ago(30m)
    GROUP BY hostname
)
SELECT REDUCE(cpu_user,
```

```
    DOUBLE '0.0',
    (s, x) -> x.value * x.value + s,
    s -> s)
from time_series_view
```

Example

Construct a time series of CPU utilization of a host and determine the fraction of samples that are above the CPU threshold:

```
WITH time_series_view AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, ROUND(measure_value::double,2)),
            SEQUENCE(ago(15m), ago(1m), 10s)) AS cpu_user
    FROM sample.DevOps
    WHERE hostname = 'host-Hovjv' and measure_name = 'cpu_utilization'
        AND time > ago(30m)
    GROUP BY hostname
)
SELECT ROUND(
    REDUCE(cpu_user,
      -- initial state
      CAST(ROW(0, 0) AS ROW(count_high BIGINT, count_total BIGINT)),
      -- function to count the total points and points above a certain threshold
      (s, x) -> CAST(ROW(s.count_high + IF(x.value > 70.0, 1, 0), s.count_total + 1) AS
 ROW(count_high BIGINT, count_total BIGINT)),
      -- output function converting the counts to fraction above threshold
      s -> IF(s.count_total = 0, NULL, CAST(s.count_high AS DOUBLE) / s.count_total)),
    4) AS fraction_cpu_above_threshold
from time_series_view
```

SQL support

Timestream for LiveAnalytics supports some common SQL constructs. You can read more below.

Topics

- SELECT
- Subquery support
- SHOW statements
- DESCRIBE statements

- UNLOAD

SELECT

SELECT statements can be used to retrieve data from one or more tables. Timestream's query language supports the following syntax for **SELECT** statements:

```
[ WITH with_query [, ...] ]
            SELECT [ ALL | DISTINCT ] select_expr [, ...]
            [ function (expression) OVER (
            [ PARTITION BY partition_expr_list ]
            [ ORDER BY order_list ]
            [ frame_clause ] )
            [ FROM from_item [, ...] ]
            [ WHERE condition ]
            [ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
            [ HAVING condition]
            [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
            [ ORDER BY order_list ]
            [ LIMIT [ count | ALL ] ]
```

where

- `function (expression)` is one of the supported window functions.
- `partition_expr_list` is:

  ```
  expression | column_name [, expr_list ]
  ```

- `order_list` is:

  ```
  expression | column_name [ ASC | DESC ]
  [ NULLS FIRST | NULLS LAST ]
  [, order_list ]
  ```

- `frame_clause` is:

  ```
  ROWS | RANGE
  { UNBOUNDED PRECEDING | expression PRECEDING | CURRENT ROW } |
  {BETWEEN
  { UNBOUNDED PRECEDING | expression { PRECEDING | FOLLOWING } |
  CURRENT ROW}
  ```

```
AND
{ UNBOUNDED FOLLOWING | expression { PRECEDING | FOLLOWING } |
CURRENT ROW }}
```

- `from_item` is one of:

```
table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
from_item join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]
```

- `join_type` is one of:

```
[ INNER ] JOIN
LEFT [ OUTER ] JOIN
RIGHT [ OUTER ] JOIN
FULL [ OUTER ] JOIN
```

- `grouping_element` is one of:

```
()
expression
```

Subquery support

Timestream supports subqueries in `EXISTS` and `IN` predicates. The `EXISTS` predicate determines if a subquery returns any rows. The `IN` predicate determines if values produced by the subquery match the values or expression of in IN clause. The Timestream query language supports correlated and other subqueries.

```
SELECT t.c1
FROM (VALUES 1, 2, 3, 4, 5) AS t(c1)
WHERE EXISTS
(SELECT t.c2
 FROM (VALUES 1, 2, 3) AS t(c2)
 WHERE t.c1= t.c2
)
ORDER BY t.c1
```

c1
1

c1
2
3

```
SELECT t.c1
FROM (VALUES 1, 2, 3, 4, 5) AS t(c1)
WHERE t.c1 IN
(SELECT t.c2
 FROM (VALUES 2, 3, 4) AS t(c2)
)
ORDER BY t.c1
```

c1
2
3
4

SHOW statements

You can view all the databases in an account by using the `SHOW DATABASES` statement. The syntax is as follows:

```
SHOW DATABASES [LIKE pattern]
```

where the `LIKE` clause can be used to filter database names.

You can view all the tables in an account by using the `SHOW TABLES` statement. The syntax is as follows:

```
SHOW TABLES [FROM database] [LIKE pattern]
```

where the `FROM` clause can be used to filter database names and the `LIKE` clause can be used to filter table names.

You can view all the measures for a table by using the `SHOW MEASURES` statement. The syntax is as follows:

```
SHOW MEASURES FROM database.table [LIKE pattern]
```

where the `FROM` clause will be used to specify the database and table name and the `LIKE` clause can be used to filter measure names.

DESCRIBE statements

You can view the metadata for a table by using the `DESCRIBE` statement. The syntax is as follows:

```
DESCRIBE database.table
```

where `table` contains the table name. The describe statement returns the column names and data types for the table.

UNLOAD

Timestream for LiveAnalytics supports an `UNLOAD` command as an extension to its SQL support. Data types supported by `UNLOAD` are described in Supported data types. The `time` and unknown types do not apply to `UNLOAD`.

```
UNLOAD (SELECT statement)
 TO 's3://bucket-name/folder'
 WITH ( option = expression [, ...] )
```

where option is

```
{ partitioned_by = ARRAY[ col_name[,…] ]
 | format = [ '{ CSV | PARQUET }' ]
 | compression = [ '{ GZIP | NONE }' ]
 | encryption = [ '{ SSE_KMS | SSE_S3 }' ]
 | kms_key = '<string>'
 | field_delimiter ='<character>'
 | escaped_by = '<character>'
 | include_header = ['{true, false}']
 | max_file_size = '<value>'
}
```

SELECT statement

The query statement used to select and retrieve data from one or more Timestream for LiveAnalytics tables.

```
(SELECT column 1, column 2, column 3 from database.table
      where measure_name = "ABC" and timestamp between ago (1d) and now() )
```

TO clause

```
TO 's3://bucket-name/folder'
```

or

```
TO 's3://access-point-alias/folder'
```

The `TO` clause in the `UNLOAD` statement specifies the destination for the output of the query results. You need to provide the full path, including either Amazon S3 bucket-name or Amazon S3 access-point-alias with folder location on Amazon S3 where Timestream for LiveAnalytics writes the output file objects. The S3 bucket should be owned by the same account and in the same region. In addition to the query result set, Timestream for LiveAnalytics writes the manifest and metadata files to specified destination folder.

PARTITIONED_BY clause

```
partitioned_by = ARRAY [col_name[,…] , (default: none)
```

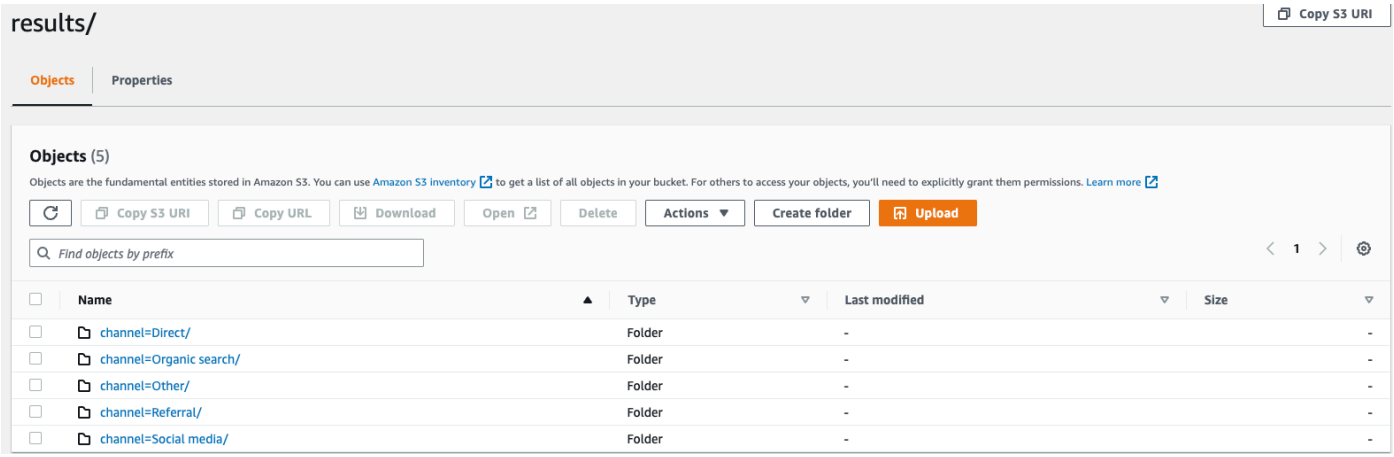
The `partitioned_by` clause is used in queries to group and analyze data at a granular level. When you export your query results to the S3 bucket, you can choose to partition the data based on one or more columns in the select query. When partitioning the data, the exported data is divided into subsets based on the partition column and each subset is stored in a separate folder. Within the results folder that contains your exported data, a sub-folder `folder/results/partition column = partition value/` is automatically created. However, note that partitioned columns are not included in the output file.

`partitioned_by` is not a mandatory clause in the syntax. If you choose to export the data without any partitioning, you can exclude the clause in the syntax.

Example

Assuming you are monitoring clickstream data of your website and have 5 channels of traffic namely `direct`, `Social Media`, `Organic Search`, `Other`, and `Referral`. When exporting the data, you can choose to partition the data using the column `Channel`. Within your data folder, `s3://bucketname/results`, you will have five folders each with their respective channel name, for instance, `s3://bucketname/results/channel=Social Media/`. Within this folder you will find the data of all the customers that landed on your website through the `Social Media` channel. Similarly, you will have other folders for the remaining channels.

Exported data partitioned by Channel column



FORMAT

```
format = [ '{ CSV | PARQUET }' , default: CSV
```

The keywords to specify the format of the query results written to your S3 bucket. You can export the data either as a comma separated value (CSV) using a comma (,) as the default delimiter or in the Apache Parquet format, an efficient open columnar storage format for analytics.

COMPRESSION

```
compression = [ '{ GZIP | NONE }' ], default: GZIP
```

You can compress the exported data using compression algorithm GZIP or have it uncompressed by specifying the `NONE` option.

ENCRYPTION

```
encryption = [ '{ SSE_KMS | SSE_S3 }' ], default: SSE_S3
```

The output files on Amazon S3 are encrypted using your selected encryption option. In addition to your data, the manifest and metadata files are also encrypted based on your selected encryption option. We currently support SSE_S3 and SSE_KMS encryption. SSE_S3 is a server-side encryption with Amazon S3 encrypting the data using 256-bit advanced encryption standard (AES) encryption. SSE_KMS is a server-side encryption to encrypt data using customer-managed keys.

KMS_KEY

```
kms_key = '<string>'
```

KMS Key is a customer-defined key to encrypt exported query results. KMS Key is securely managed by AWS Key Management Service (AWS KMS) and used to encrypt data files on Amazon S3.

FIELD_DELIMITER

```
field_delimiter ='<character>' , default: (,)
```

When exporting the data in CSV format, this field specifies a single ASCII character that is used to separate fields in the output file, such as pipe character (|), a comma (,), or tab (/t). The default delimiter for CSV files is a comma character. If a value in your data contains the chosen delimiter, the delimiter will be quoted with a quote character. For instance, if the value in your data contains `Time,stream`, then this value will be quoted as `"Time,stream"` in the exported data. The quote character used by Timestream for LiveAnalytics is double quotes (").

Avoid specifying the carriage return character (ASCII 13, hex `0D`, text '\r') or the line break character (ASCII 10, hex 0A, text '\n') as the `FIELD_DELIMITER` if you want to include headers in the CSV, since that will prevent many parsers from being able to parse the headers correctly in the resulting CSV output.

ESCAPED_BY

```
escaped_by = '<character>', default: (\)
```

When exporting the data in CSV format, this field specifies the character that should be treated as an escape character in the data file written to S3 bucket. Escaping happens in the following scenarios:

1. If the value itself contains the quote character (") then it will be escaped using an escape character. For example, if the value is `Time"stream`, where (\) is the configured escape character, then it will be escaped as `Time\"stream`.
2. If the value contains the configured escape character, it will be escaped. For example, if the value is `Time\stream`, then it will be escaped as `Time\\stream`.

> **Note**
>
> If the exported output contains complex data type in the like Arrays, Rows or Timeseries, it will be serialized as a JSON string. Following is an example.

Data type	Actual value	How the value is escaped in CSV format [serialized JSON string]
Array	`[ 23,24,25 ]`	`"[23,24,25]"`
Row	`( x=23.0, y=hello )`	`"{\"x\":23.0,\"y\":\"hello\"}"`
Timeseries	`[ ( time=1970-01-01 00:00:00.000000010 , value=100.0 ), ( time=1970-01-01 00:00:00.000000012, value=120.0 ) ]`	`"[{\"time\":\"1970-01-01 00:00:00.000000010Z\",\"value\":100.0},{\"time\":\"1970-01-01 00:00:00.000000012Z\",\"value\":120.0}]"`

INCLUDE_HEADER

```
include_header = 'true' , default: 'false'
```

When exporting the data in CSV format, this field lets you include column names as the first row of the exported CSV data files.

The accepted values are 'true' and 'false' and the default value is 'false'. Text transformation options such as `escaped_by` and `field_delimiter` apply to headers as well.

> **Note**
>
> When including headers, it is important that you not select a carriage return character (ASCII 13, hex 0D, text '\r') or a line break character (ASCII 10, hex 0A, text '\n') as the `FIELD_DELIMITER`, since that will prevent many parsers from being able to parse the headers correctly in the resulting CSV output.

MAX_FILE_SIZE

```
max_file_size = 'X[MB|GB]' , default: '78GB'
```

This field specifies the maximum size of the files that the `UNLOAD` statement creates in Amazon S3. The `UNLOAD` statement can create multiple files but the maximum size of each file written to Amazon S3 will be approximately what is specified in this field.

The value of the field must be between 16 MB and 78 GB, inclusive. You can specify it in integer such as `12GB`, or in decimals such as `0.5GB` or `24.7MB`. The default value is 78 GB.

The actual file size is approximated when the file is being written, so the actual maximum size may not be exactly equal to the number you specify.

Logical operators

Timestream for LiveAnalytics supports the following logical operators.

Operator	Description	Example
AND	True if both values are true	a AND b
OR	True if either value is true	a OR b
NOT	True if the value is false	NOT a

- The result of an AND comparison may be NULL if one or both sides of the expression are NULL.
- If at least one side of an AND operator is FALSE the expression evaluates to FALSE.
- The result of an OR comparison may be NULL if one or both sides of the expression are NULL.
- If at least one side of an OR operator is TRUE the expression evaluates to TRUE.
- The logical complement of NULL is NULL.

The following truth table demonstrates the handling of NULL in AND and OR:

A	B	A and b	A or b
null	null	null	null
false	null	false	null
null	false	false	null
true	null	null	true
null	true	null	true
false	false	false	false
true	false	false	true
false	true	false	true
true	true	true	true

The following truth table demonstrates the handling of NULL in NOT:

A	Not a
null	null
true	false
false	true

Comparison operators

Timestream for LiveAnalytics supports the following comparison operators.

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal
<>	Not equal
!=	Not equal

Note

- The `BETWEEN` operator tests if a value is within a specified range. The syntax is as follows:

  ```
  BETWEEN min AND max
  ```

  The presence of `NULL` in a `BETWEEN` or `NOT BETWEEN` statement will result in the statement evaluating to `NULL`.
- `IS NULL` and `IS NOT NULL` operators test whether a value is null (undefined). Using `NULL` with `IS NULL` evaluates to true.
- In SQL, a `NULL` value signifies an unknown value.

Comparison functions

Timestream for LiveAnalytics supports the following comparison functions.

Topics

- greatest()
- least()
- ALL(), ANY() and SOME()

greatest()

The **greatest()** function returns the largest of the provided values. It returns NULL if any of the provided values are NULL. The syntax is as follows.

```
greatest(value1, value2, ..., valueN)
```

least()

The **least()** function returns the smallest of the provided values. It returns NULL if any of the provided values are NULL. The syntax is as follows.

```
least(value1, value2, ..., valueN)
```

ALL(), ANY() and SOME()

The ALL, ANY and SOME quantifiers can be used together with comparison operators in the following way.

Expression	Meaning
A = ALL(...)	Evaluates to true when A is equal to all values.
A <> ALL(...)	Evaluates to true when A does not match any value.
A < ALL(...)	Evaluates to true when A is smaller than the smallest value.
A = ANY(...)	Evaluates to true when A is equal to any of the values.
A <> ANY(...)	Evaluates to true when A does not match one or more values.

Expression	Meaning
A < ANY(...)	Evaluates to true when A is smaller than the biggest value.

Examples and usage notes

> **Note**
>
> When using `ALL`, `ANY` or `SOME`, the keyword `VALUES` should be used if the comparison values are a list of literals.

Example: `ANY()`

An example of `ANY()` in a query statement as follows.

```
SELECT 11.7 = ANY (VALUES 12.0, 13.5, 11.7)
```

An alternative syntax for the same operation is as follows.

```
SELECT 11.7 = ANY (SELECT 12.0 UNION ALL SELECT 13.5 UNION ALL SELECT 11.7)
```

In this case, `ANY()` evaluates to `True`.

Example: `ALL()`

An example of `ALL()` in a query statement as follows.

```
SELECT 17 < ALL (VALUES 19, 20, 15);
```

An alternative syntax for the same operation is as follows.

```
SELECT 17 < ALL (SELECT 19 UNION ALL SELECT 20 UNION ALL SELECT 15);
```

In this case, `ALL()` evaluates to `False`.

Example: `SOME()`

An example of `SOME()` in a query statement as follows.

```
SELECT 50 >= SOME (VALUES 53, 77, 27);
```

An alternative syntax for the same operation is as follows.

```
SELECT 50 >= SOME (SELECT 53 UNION ALL SELECT 77 UNION ALL SELECT 27);
```

In this case, `SOME()` evaluates to `True`.

Conditional expressions

Timestream for LiveAnalytics supports the following conditional expressions.

Topics

- The CASE statement
- The IF statement
- The COALESCE statement
- The NULLIF statement
- The TRY statement

The CASE statement

The **CASE** statement searches each value expression from left to right until it finds one that equals `expression`. If it finds a match, the result for the matching value is returned. If no match is found, the result from the `ELSE` clause is returned if it exists; otherwise `null` is returned. The syntax is as follows:

```
CASE expression
    WHEN value THEN result
    [ WHEN ... ]
    [ ELSE result ]
END
```

Timestream also supports the following syntax for **CASE** statements. In this syntax, the "searched" form evaluates each boolean condition from left to right until one is `true` and returns the matching result. If no conditions are `true`, the result from the `ELSE` clause is returned if it exists; otherwise `null` is returned. See below for the alternate syntax:

```
CASE
    WHEN condition THEN result
    [ WHEN ... ]
    [ ELSE result ]
END
```

The IF statement

The **IF** statement evaluates a condition to be true or false and returns the appropriate value. Timestream supports the following two syntax representations for **IF**:

```
if(condition, true_value)
```

This syntax evaluates and returns `true_value` if condition is `true`; otherwise `null` is returned and `true_value` is not evaluated.

```
if(condition, true_value, false_value)
```

This syntax evaluates and returns `true_value` if condition is `true`, otherwise evaluates and returns `false_value`.

Examples

```
SELECT
  if(true, 'example 1'),
  if(false, 'example 2'),
  if(true, 'example 3 true', 'example 3 false'),
  if(false, 'example 4 true', 'example 4 false')
```

_col0	_col1	_col2	_col3
example 1	-	example 3 true	example 4 false
	null		

The COALESCE statement

COALESCE returns the first non-null value in an argument list. The syntax is as follows:

```
coalesce(value1, value2[,...])
```

The NULLIF statement

The **IF** statement evaluates a condition to be true or false and returns the appropriate value. Timestream supports the following two syntax representations for **IF**:

NULLIF returns null if `value1` equals `value2`; otherwise it returns `value1`. The syntax is as follows:

```
nullif(value1, value2)
```

The TRY statement

The **TRY** function evaluates an expression and handles certain types of errors by returning `null`. The syntax is as follows:

```
try(expression)
```

Conversion functions

Timestream for LiveAnalytics supports the following conversion functions.

Topics

- cast()
- try_cast()

cast()

The syntax of the cast function to explicitly cast a value as a type is as follows.

```
cast(value AS type)
```

try_cast()

Timestream for LiveAnalytics also supports the try_cast function that is similar to cast but returns null if cast fails. The syntax is as follows.

```
try_cast(value AS type)
```

Mathematical operators

Timestream for LiveAnalytics supports the following mathematical operators.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (integer division performs truncation)
%	Modulus (remainder)

Mathematical functions

Timestream for LiveAnalytics supports the following mathematical functions.

Function	Output data type	Description
abs(x)	[same as input]	Returns the absolute value of x.
cbrt(x)	double	Returns the cube root of x.
ceiling(x) or ceil(x)	[same as input]	Returns x rounded up to the nearest integer.
degrees(x)	double	Converts angle x in radians to degrees.
e()	double	Returns the constant Euler's number.

Function	Output data type	Description
exp(x)	double	Returns Euler's number raised to the power of x.
floor(x)	[same as input]	Returns x rounded down to the nearest integer.
from_base(string,radix)	bigint	Returns the value of string interpreted as a base-radix number.
ln(x)	double	Returns the natural logarithm of x.
log2(x)	double	Returns the base 2 logarithm of x.
log10(x)	double	Returns the base 10 logarithm of x.
mod(n,m)	[same as input]	Returns the modulus (remainder) of n divided by m.
pi()	double	Returns the constant Pi.
pow(x, p) or power(x, p)	double	Returns x raised to the power of p.
radians(x)	double	Converts angle x in degrees to radians.
rand() or random()	double	Returns a pseudo-random value in the range 0.0 1.0.
random(n)	[same as input]	Returns a pseudo-random number between 0 and n (exclusive).

Function	Output data type	Description
round(x)	[same as input]	Returns x rounded to the nearest integer.
round(x,d)	[same as input]	Returns x rounded to d decimal places.
sign(x)	[same as input]	Returns the signum function of x, that is: • 0 if the argument is 0 • 1 if the argument is greater than 0 • -1 if the argument is less than 0. For double arguments, the function additionally returns: • NaN if the argument is NaN • 1 if the argument is +Infinity • -1 if the argument is -Infinity.
sqrt(x)	double	Returns the square root of x.
to_base(x, radix)	varchar	Returns the base-radi x representation of x.
truncate(x)	double	Returns x rounded to integer by dropping digits after decimal point.
acos(x)	double	Returns the arc cosine of x.
asin(x)	double	Returns the arc sine of x.

Function	Output data type	Description
atan(x)	double	Returns the arc tangent of x.
atan2(y, x)	double	Returns the arc tangent of y / x.
cos(x)	double	Returns the cosine of x.
cosh(x)	double	Returns the hyperbolic cosine of x.
sin(x)	double	Returns the sine of x.
tan(x)	double	Returns the tangent of x.
tanh(x)	double	Returns the hyperbolic tangent of x.
infinity()	double	Returns the constant representing positive infinity.
is_finite(x)	boolean	Determine if x is finite.
is_infinite(x)	boolean	Determine if x is infinite.
is_nan(x)	boolean	Determine if x is not-a-number.
nan()	double	Returns the constant representing not-a-number.

String operators

Timestream for LiveAnalytics supports the `||` operator for concatenating one or more strings.

String functions

> **Note**
>
> The input data type of these functions is assumed to be varchar unless otherwise specified.

Function	Output data type	Description
chr(n)	varchar	Returns the Unicode code point n as a varchar.
codepoint(x)	integer	Returns the Unicode code point of the only character of str.
concat(x1, ..., xN)	varchar	Returns the concatenation of x1, x2, ..., xN.
hamming_distance(x1,x2)	bigint	Returns the Hamming distance of x1 and x2, i.e. the number of positions at which the corresponding character s are different. Note that the two varchar inputs must have the same length.
length(x)	bigint	Returns the length of x in characters.
levenshtein_distance(x1, x2)	bigint	Returns the Levenshtein edit distance of x1 and x2, i.e. the minimum number of single-character edits (insertions, deletions or substitutions) needed to change x1 into x2.
lower(x)	varchar	Converts x to lowercase.

Function	Output data type	Description
lpad(x1, bigint size, x2)	varchar	Left pads x1 to size character s with x2. If size is less than the length of x1, the result is truncated to size characters. size must not be negative and x2 must be non-empty.
ltrim(x)	varchar	Removes leading whitespace from x.
replace(x1, x2)	varchar	Removes all instances of x2 from x1.
replace(x1, x2, x3)	varchar	Replaces all instances of x2 with x3 in x1.
Reverse(x)	varchar	Returns x with the characters in reverse order.
rpad(x1, bigint size, x2)	varchar	Right pads x1 to size characters with x2. If size is less than the length of x1, the result is truncated to size characters. size must not be negative and x2 must be non-empty.
rtrim(x)	varchar	Removes trailing whitespace from x.
split(x1, x2)	array(varchar)	Splits x1 on delimiter x2 and returns an array.

Function	Output data type	Description
split(x1, x2, bigint limit)	array(varchar)	Splits x1 on delimiter x2 and returns an array. The last element in the array always contain everything left in the x1. limit must be a positive number.
split_part(x1, x2, bigint pos)	varchar	Splits x1 on delimiter x2 and returns the varchar field at pos. Field indexes start with 1. If pos is larger than the number of fields, then null is returned.
strpos(x1, x2)	bigint	Returns the starting position of the first instance of x2 in x1. Positions start with 1. If not found, 0 is returned.
strpos(x1, x2,bigint instance)	bigint	Returns the position of the Nth instance of x2 in x1. Instance must be a positive number. Positions start with 1. If not found, 0 is returned.
strrpos(x1, x2)	bigint	Returns the starting position of the last instance of x2 in x1. Positions start with 1. If not found, 0 is returned.
strrpos(x1, x2, bigint instance)	bigint	Returns the position of the Nth instance of x2 in x1 starting from the end of x1. instance must be a positive number. Positions start with 1. If not found, 0 is returned.

Function	Output data type	Description
position(x2 IN x1)	bigint	Returns the starting position of the first instance of x2 in x1. Positions start with 1. If not found, 0 is returned.
substr(x, bigint start)	varchar	Returns the rest of x from the starting position start. Positions start with 1. A negative starting position is interpreted as being relative to the end of x.
substr(x, bigint start, bigint len)	varchar	Returns a substring from x of length len from the starting position start. Positions start with 1. A negative starting position is interpreted as being relative to the end of x.
trim(x)	varchar	Removes leading and trailing whitespace from x.
upper(x)	varchar	Converts x to uppercase.

Array operators

Timestream for LiveAnalytics supports the following array operators.

Operator	Description
[]	Access an element of an array where the first index starts at 1.
||	Concatenate an array with another array or element of the same type.

Array functions

Timestream for LiveAnalytics supports the following array functions.

Function	Output data type	Description
array_distinct(x)	array	Remove duplicate values from the array x. `SELECT array_distinct(ARRAY[1,2,2,3])` Example result: `[ 1,2,3 ]`
array_intersect(x, y)	array	Returns an array of the elements in the intersection of x and y, without duplicates. `SELECT array_intersect(ARRAY[1,2,3], ARRAY[3,4,5])` Example result: `[ 3 ]`
array_union(x, y)	array	Returns an array of the elements in the union of x and y, without duplicates. `SELECT array_union(ARRAY[1,2,3], ARRAY[3,4,5])` Example result: `[ 1,2,3,4,5 ]`
array_except(x, y)	array	Returns an array of elements in x but not in y, without duplicates.

Function	Output data type	Description
		`SELECT array_exc ept(ARRAY[1,2,3], ARRAY[3,4,5])` Example result: `[ 1,2 ]`
array_join(x, delimiter, null_replacement)	varchar	Concatenates the elements of the given array using the delimiter and an optional string to replace nulls. `SELECT array_joi n(ARRAY[1,2,3], ';', '')` Example result: `1;2;3`
array_max(x)	same as array elements	Returns the maximum value of input array. `SELECT array_max (ARRAY[1,2,3])` Example result: 3
array_min(x)	same as array elements	Returns the minimum value of input array. `SELECT array_min (ARRAY[1,2,3])` Example result: 1

Function	Output data type	Description
array_position(x, element)	bigint	Returns the position of the first occurrence of the element in array x (or 0 if not found). `SELECT array_position(ARRAY[3,4,5,9], 5)` Example result: 3
array_remove(x, element)	array	Remove all elements that equal element from array x. `SELECT array_remove(ARRAY[3,4,5,9], 4)` Example result: `[ 3,5,9 ]`
array_sort(x)	array	Sorts and returns the array x. The elements of x must be orderable. Null elements will be placed at the end of the returned array. `SELECT array_sort(ARRAY[6,8,2,9,3])` Example result: `[ 2,3,6,8,9 ]`

Function	Output data type	Description
arrays_overlap(x, y)	boolean	Tests if arrays x and y have any non-null elements in common. Returns null if there are no non-null elements in common but either array contains null. `SELECT arrays_overlap(ARRAY[6,8,2,9,3], ARRAY[6,8])` Example result: `true`
cardinality(x)	bigint	Returns the size of the array x. `SELECT cardinality(ARRAY[6,8,2,9,3])` Example result: 5
concat(array1, array2, ..., arrayN)	array	Concatenates the arrays array1, array2, ..., arrayN. `SELECT concat(ARRAY[6,8,2,9,3], ARRAY[11,32], ARRAY[6,8,2,0,14])` Example result: `[ 6,8,2,9,3,11,32,6,8,2,0,14 ]`

Function	Output data type	Description
element_at(array(E), index)	E	Returns element of array at given index. If index < 0, element_at accesses elements from the last to the first. `SELECT element_at(ARRAY[6,8,2,9,3], 1)` Example result: 6
repeat(element, count)	array	Repeat element for count times. `SELECT repeat(1, 3)` Example result: `[ 1,1,1 ]`
reverse(x)	array	Returns an array which has the reversed order of array x. `SELECT reverse(ARRAY[6,8,2,9,3])` Example result: `[ 3,9,2,8,6 ]`

Function	Output data type	Description

sequence(start, stop)	array(bigint)	Generate a sequence of integers from start to stop, incrementing by 1 if start is less than or equal to stop, otherwise -1.
		`SELECT sequence(3, 8)`
		Example result:
		`[ 3,4,5,6,7,8 ]`

sequence(start, stop, step)	array(bigint)	Generate a sequence of integers from start to stop, incrementing by step.
		`SELECT sequence(3, 15, 2)`
		Example result:
		`[ 3,5,7,9,11,13,15 ]`

Function	Output data type	Description

sequence(start, stop)	array(timestamp)	Generate a sequence of timestamps from start date to stop date, incrementing by 1 day.
		
		`SELECT sequence( '2023-04-02 19:26:12.941000000', '2023-04-06 19:26:12.941000000', 1d)`
		
		Example result:
		`[ 2023-04-02 19:26:12.941000000 ,2023-04-03 19:26:12.941000000 ,2023-04-04 19:26:12.941000000 ,2023-04-05 19:26:12.941000000 ,2023-04-06 19:26:12.941000000 ]`

Function	Output data type	Description

sequence(start, stop, step)	array(timestamp)	Generate a sequence of timestamps from start to stop, incrementing by step. The data type of step is interval.
		
		`SELECT sequence( '2023-04-02 19:26:12.941000000', '2023-04-10 19:26:12.941000000', 2d)`
		
		Example result:
		`[ 2023-04-02 19:26:12.941000000 ,2023-04-04 19:26:12.941000000 ,2023-04-06 19:26:12.941000000 ,2023-04-08 19:26:12.941000000 ,2023-04-10 19:26:12.941000000 ]`

shuffle(x)	array	Generate a random permutation of the given array x.
		
		`SELECT shuffle(ARRAY[6,8,2,9,3])`
		
		Example result:
		`[ 6,3,2,9,8 ]`

Function	Output data type	Description
slice(x, start, length)	array	Subsets array x starting from index start (or starting from the end if start is negative) with a length of length. `SELECT slice(ARRAY[6,8,2,9,3], 1, 3)` Example result: `[ 6,8,2 ]`
zip(array1, array2[, ...])	array(row)	Merges the given arrays, element-wise, into a single array of rows. If the arguments have an uneven length, missing values are filled with NULL. `SELECT zip(ARRAY[6,8,2,9,3], ARRAY[15, 24])` Example result: `[ ( 6, 15 ),( 8, 24 ),( 2, - ),( 9, - ),( 3, - ) ]`

Bitwise functions

Timestream for LiveAnalytics supports the following bitwise functions.

Function	Output data type	Description
bit_count(bigint, bigint)	bigint (two's complement)	Returns the count of bits in the first bigint parameter where the second parameter

Function	Output data type	Description

		is a bit signed integer such as 8 or 64.
		
		`SELECT bit_count(19, 8)`
		
		Example result: 3
		
		`SELECT bit_count(19, 2)`
		
		Example result: `Number must be represent able with the bits specified. 19 can not be represented with 2 bits`

bitwise_and(bigint, bigint)	bigint (two's complement)	Returns the bitwise AND of the bigint parameters. `SELECT bitwise_and(12, 7)` Example result: 4
bitwise_not(bigint)	bigint (two's complement)	Returns the bitwise NOT of the bigint parameter. `SELECT bitwise_not(12)` Example result: `-13`

Function	Output data type	Description
bitwise_or(bigint, bigint)	bigint (two's complement)	Returns the bitwise OR of the bigint parameters. `SELECT bitwise_or(12, 7)` Example result: 15
bitwise_xor(bigint, bigint)	bigint (two's complement)	Returns the bitwise XOR of the bigint parameters. `SELECT bitwise_xor(12, 7)` Example result: 11

Regular expression functions

The regular expression functions in Timestream for LiveAnalytics support the Java pattern syntax. Timestream for LiveAnalytics supports the following regular expression functions.

Function	Output data type	Description
regexp_extract_all(string, pattern)	array(varchar)	Returns the substring(s) matched by the regular expression pattern in string. `SELECT regexp_extract_all('example expect complex', 'ex\w')` Example result: `[ exa,exp ]`

Function	Output data type	Description
regexp_extract_all(string, pattern, group)	array(varchar)	Finds all occurrences of the regular expression pattern in string and returns the capturing group number group. `SELECT regexp_extract_all('example expect complex', '(ex)(\w)', 2)` Example result: `[ a,p ]`
regexp_extract(string, pattern)	varchar	Returns the first substring matched by the regular expression pattern in string. `SELECT regexp_extract('example expect', 'ex\w')` Example result: `exa`
regexp_extract(string, pattern, group)	varchar	Finds the first occurrence of the regular expression pattern in string and returns the capturing group number group. `SELECT regexp_extract('example expect', '(ex)(\w)', 2)` Example result: a

Function	Output data type	Description
regexp_like(string, pattern)	boolean	Evaluates the regular expression pattern and determines if it is contained within string. This function is similar to the LIKE operator, except that the pattern only needs to be contained within string, rather than needing to match all of string. In other words, this performs a contains operation rather than a match operation. You can match the entire string by anchoring the pattern using ^ and $. `SELECT regexp_like('example', 'ex')` Example result: `true`
regexp_replace(string, pattern)	varchar	Removes every instance of the substring matched by the regular expression pattern from string. `SELECT regexp_replace('example expect', 'expect')` Example result: `example`

Function	Output data type	Description
regexp_replace(string, pattern, replacement)	varchar	Replaces every instance of the substring matched by the regex pattern in string with replacement. Capturing groups can be referenced in replacement using $g for a numbered group or ${name} for a named group. A dollar sign ($) may be included in the replacement by escaping it with a backslash (\$). `SELECT regexp_replace('example expect', 'expect', 'surprise')` Example result: `example surprise`

Function	Output data type	Description
regexp_replace(string, pattern, function)	varchar	Replaces every instance of the substring matched by the regular expression pattern in string using function. The lambda expression function is invoked for each match with the capturing groups passed as an array. Capturing group numbers start at one; there is no group for the entire match (if you need this, surround the entire expression with parenthesis). `SELECT regexp_replace('example', '(\w)', x -> upper(x[1]))` Example result: `EXAMPLE`
regexp_split(string, pattern)	array(varchar)	Splits string using the regular expression pattern and returns an array. Trailing empty strings are preserved. `SELECT regexp_split('example', 'x')` Example result: `[ e,ample ]`

Date / time operators

> **Note**
>
> Timestream for LiveAnalytics does not support negative time values. Any operation resulting in negative time results in error.

Timestream for LiveAnalytics supports the following operations on `timestamps`, `dates`, and `intervals`.

Operator	Description
+	Addition
-	Subtraction

Topics

- Operations
- Addition
- Subtraction

Operations

The result type of an operation is based on the operands. Interval literals such as `1day` and `3s` can be used.

```
SELECT date '2022-05-21' + interval '2' day
```

```
SELECT date '2022-05-21' + 2d
```

```
SELECT date '2022-05-21' + 2day
```

Example result for each: `2022-05-23`

Interval units include `second`, `minute`, `hour`, `day`, `week`, `month`, and `year`. But in some cases not all are applicable. For example seconds, minutes, and hours can not be added to or subtracted from a date.

```
SELECT interval '4' year + interval '2' month
```

Example result: `4-2`

```
SELECT typeof(interval '4' year + interval '2' month)
```

Example result: `interval year to month`

Result type of interval operations may be `'interval year to month'` or `'interval day to second'` depending on the operands. Intervals can be added to or subtracted from `dates` and `timestamps`. But a `date` or `timestamp` cannot be added to or subtracted from a `date` or `timestamp`. To find intervals or durations related to dates or timestamps, see `date_diff` and related functions in Interval and duration.

Addition

Example

```
SELECT date '2022-05-21' + interval '2' day
```

Example result: `2022-05-23`

Example

```
SELECT typeof(date '2022-05-21' + interval '2' day)
```

Example result: `date`

Example

```
SELECT interval '2' year + interval '4' month
```

Example result: `2-4`

Example

```
SELECT typeof(interval '2' year + interval '4' month)
```

Example result: `interval year to month`

Subtraction

Example

```
SELECT timestamp '2022-06-17 01:00' - interval '7' hour
```

Example result: `2022-06-16 18:00:00.000000000`

Example

```
SELECT typeof(timestamp '2022-06-17 01:00' - interval '7' hour)
```

Example result: `timestamp`

Example

```
SELECT interval '6' day - interval '4' hour
```

Example result: `5 20:00:00.000000000`

Example

```
SELECT typeof(interval '6' day - interval '4' hour)
```

Example result: `interval day to second`

Date / time functions

> **Note**
>
> Timestream for LiveAnalytics does not support negative time values. Any operation resulting in negative time results in error.

Timestream for LiveAnalytics uses UTC timezone for date and time. Timestream supports the following functions for date and time.

Topics

- General and conversion
- Interval and duration
- Formatting and parsing
- Extraction

General and conversion

Timestream for LiveAnalytics supports the following general and conversion functions for date and time.

Function	Output data type	Description
current_date	date	Returns current date in UTC. No parentheses used. `SELECT current_date` Example result: `2022-07-07` **Note** This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords.
current_time	time	Returns current time in UTC. No parentheses used. `SELECT current_time`

Function	Output data type	Description
		Example result: `17:41:52.827000000` **Note** This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords.
current_timestamp or now()	timestamp	Returns current timestamp in UTC. `SELECT current_timestamp` Example result: `2022-07-07 17:42:32.939000000` **Note** This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords.

Function	Output data type	Description
current_timezone()	varchar The value will be 'UTC.'	Timestream uses UTC timezone for date and time. `SELECT current_timezone()` Example result: UTC
date(varchar(x)), date(timestamp)	date	`SELECT date(TIMESTAMP '2022-07-07 17:44:43.771000000')` Example result: `2022-07-07`
last_day_of_month(timestamp), last_day_of_month(date)	date	`SELECT last_day_of_month(TIMESTAMP '2022-07-07 17:44:43.771000000')` Example result: `2022-07-31`
from_iso8601_timestamp(string)	timestamp	Parses the ISO 8601 timestamp into internal timestamp format. `SELECT from_iso8601_timestamp('2022-06-17T08:04:05.000000000+05:00')` Example result: `2022-06-17 03:04:05.000000000`

Function	Output data type	Description
from_iso8601_date(string)	date	Parses the ISO 8601 date string into internal timestamp format for UTC 00:00:00 of the specified date. `SELECT from_iso8601_date('2022-07-17')` Example result: `2022-07-17`
to_iso8601(timestamp), to_iso8601(date)	varchar	Returns an ISO 8601 formatted string for the input. `SELECT to_iso8601(from_iso8601_date('2022-06-17'))` Example result: `2022-06-17`
from_milliseconds(bigint)	timestamp	`SELECT from_milliseconds(1)` Example result: `1970-01-01 00:00:00.001000000`

Function	Output data type	Description
from_nanoseconds(bigint)	timestamp	`select from_nanoseconds(300000001)` Example result: `1970-01-01 00:00:00.300000001`
from_unixtime(double)	timestamp	Returns a timestamp which corresponds to the provided unixtime. `SELECT from_unixtime(1)` Example result: `1970-01-01 00:00:01.000000000`
localtime	time	Returns current time in UTC. No parentheses used. `SELECT localtime` Example result: `17:58:22.654000000` **Note** This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords.

Function	Output data type	Description
localtimestamp	timestamp	Returns current timestamp in UTC. No parentheses used. `SELECT localtimestamp` Example result: `2022-07-07 17:59:04.368000000` **Note** This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords.

to_milliseconds(interval day to second), to_milliseconds(timestamp)	bigint	`SELECT to_milliseconds(INTERVAL '2' DAY + INTERVAL '3' HOUR)`
		Example result: `183600000`
		`SELECT to_milliseconds(TIMESTAMP '2022-06-17 17:44:43.771000000')`
		Example result: `1655487883771`

Function	Output data type	Description

to_nanoseconds(interval day to second), to_nanoseconds(timestamp)	bigint	`SELECT to_nanoseconds(INTERVAL '2' DAY + INTERVAL '3' HOUR)`
		
		Example result: `183600000000000`
		
		`SELECT to_nanoseconds(TIMESTAMP '2022-06-17 17:44:43.771000678')`
		
		Example result: `1655487883771000678`

to_unixtime(timestamp)	double	Returns unixtime for the provided timestamp. `SELECT to_unixtime('2022-06-17 17:44:43.771000000')` Example result: `1.6554878837710001E9`

Function	Output data type	Description
date_trunc(unit, timestamp)	timestamp	Returns the timestamp truncated to unit, where unit is one of [second, minute, hour, day, week, month, quarter, or year]. `SELECT date_trunc('minute', TIMESTAMP '2022-06-17 17:44:43.771000000')` Example result: `2022-06-17 17:44:00.000000000`

Interval and duration

Timestream for LiveAnalytics supports the following interval and duration functions for date and time.

Function	Output data type	Description
date_add(unit, bigint, date), date_add(unit, bigint, time), date_add(varchar(x), bigint, timestamp)	timestamp	Adds a bigint of units, where unit is one of [second, minute, hour, day, week, month, quarter, or year]. `SELECT date_add('hour', 9, TIMESTAMP '2022-06-17 00:00:00')` Example result: `2022-06-17 09:00:00.000000000`

Function	Output data type	Description
date_diff(unit, date, date) , date_diff(unit, time, time) , date_diff(unit, timestamp, timestamp)	bigint	Returns a difference, where unit is one of [second, minute, hour, day, week, month, quarter, or year]. `SELECT date_diff('day', DATE '2020-03-01', DATE '2020-03-02')` Example result: `1`

parse_duration(string)	interval	Parses the input string to return an `interval` equivalent.
		`SELECT parse_duration('42.8ms')`
		Example result: `0 00:00:00.042800000`
		`SELECT typeof(parse_duration('42.8ms'))`
		Example result: `interval day to second`

Function	Output data type	Description
bin(timestamp, interval)	timestamp	Rounds down the `timestamp` parameter's integer value to the nearest multiple of the `interval` parameter's integer value. The meaning of this return value may not be obvious. It is calculated using integer arithmetic first by dividing the timestamp integer by the interval integer and then by multiplying the result by the interval integer. Keeping in mind that a timestamp specifies a UTC point in time as a number of fractions of a second elapsed since the POSIX epoch (January 1, 1970), the return value will seldom align with calendar units. For example, if you specify an interval of 30 days, all the days since the epoch are divided into 30-day increments, and the start of the most recent 30-day increment is returned, which has no relationship to calendar months. Here are some examples: `bin(TIMESTAMP '2022-06-17 10:15:20', 5m)`

Function	Output data type	Description
		==> 2022-06-17 10:15:00.000000000 bin(TIMESTAMP '2022-06-17 10:15:20', 1d) ==> 2022-06-17 00:00:00.000000000 bin(TIMESTAMP '2022-06-17 10:15:20', 10day) ==> 2022-06-17 00:00:00.000000000 bin(TIMESTAMP '2022-06-17 10:15:20', 30day) ==> 2022-05-28 00:00:00.000000000
ago(interval)	timestamp	Returns the value correspon ding to current_timestamp `interval`. `SELECT ago(1d)` Example result: `2022-07-06 21:08:53.245000000`
interval literals such as 1h, 1d, and 30m	interval	Interval literals are a convenience for parse_dur ation(string). For example, 1d is the same as `parse_duration('1d')`. This allows the use of the literals wherever an interval is used. For example, `ago(1d)` and `bin(<timestamp>, 1m)`.

Some interval literals act as shorthand for parse_duration. For example, `parse_duration('1day')`, `1day`, `parse_duration('1d')`, and `1d` each return `1 00:00:00.000000000` where the type is `interval day to second`. Space is allowed in the format provided to `parse_duration`. For example `parse_duration('1day')` also returns `00:00:00.000000000`. But `1 day` is not an interval literal.

The units related to `interval day to second` are ns, nanosecond, us, microsecond, ms, millisecond, s, second, m, minute, h, hour, d, and day.

There is also `interval year to month`. The units related to interval year to month are y, year, and month. For example, `SELECT 1year` returns `1-0`. `SELECT 12month` also returns `1-0`. `SELECT 8month` returns `0-8`.

Although the unit of `quarter` is also available for some functions such as `date_trunc` and `date_add`, `quarter` is not available as part of an interval literal.

Formatting and parsing

Timestream for LiveAnalytics supports the following formatting and parsing functions for date and time.

Function	Output data type	Description
date_format(timestamp, varchar(x))	varchar	For more information about the format specifiers used by this function, see https://trino.io/docs/current/functions/datetime.html#mysql-date-functions `SELECT date_format(TIMESTAMP '2019-10-20 10:20:20', '%Y-%m-%d %H:%i:%s')` Example result: `2019-10-20 10:20:20`

Function	Output data type	Description
date_parse(varchar(x), varchar(y))	timestamp	For more information about the format specifiers used by this function, see https://trino.io/docs/current/functions/datetime.html#mysql-date-functions `SELECT date_parse('2019-10-20 10:20:20', '%Y-%m-%d %H:%i:%s')` Example result: `2019-10-20 10:20:20.000000000`
format_datetime(timestamp, varchar(x))	varchar	For more information about the format string used by this function, see http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html `SELECT format_datetime(parse_datetime('1968-01-13 12', 'yyyy-MM-dd HH'), 'yyyy-MM-dd HH')` Example result: `1968-01-13 12`

Function	Output data type	Description
parse_datetime(varchar(x), varchar(y))	timestamp	For more information about the format string used by this function, see http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html `SELECT parse_datetime('2019-12-29 10:10 PST', 'uuuu-LL-dd HH:mm z')` Example result: `2019-12-29 18:10:00.000000000`

Extraction

Timestream for LiveAnalytics supports the following extraction functions for date and time. The extract function is the basis for the remaining convenience functions.

Function	Output data type	Description
extract	bigint	Extracts a field from a timestamp, where field is one of [YEAR, QUARTER, MONTH, WEEK, DAY, DAY_OF_MONTH, DAY_OF_WEEK, DOW, DAY_OF_YEAR, DOY, YEAR_OF_WEEK, YOW, HOUR, MINUTE, or SECOND]. `SELECT extract(YEAR FROM '2019-10-12 23:10:34.000000000')`

Function	Output data type	Description
		Example result: 2019
day(timestamp), day(date), day(interval day to second)	bigint	`SELECT day('2019-10-12 23:10:34.000000000')` Example result: 12
day_of_month(timestamp), day_of_month(date), day_of_month(interval day to second)	bigint	`SELECT day_of_month('2019-10-12 23:10:34.000000000')` Example result: 12
day_of_week(timestamp), day_of_week(date)	bigint	`SELECT day_of_week('2019-10-12 23:10:34.000000000')` Example result: 6
day_of_year(timestamp), day_of_year(date)	bigint	`SELECT day_of_year('2019-10-12 23:10:34.000000000')` Example result: 285
dow(timestamp), dow(date)	bigint	Alias for day_of_week
doy(timestamp), doy(date)	bigint	Alias for day_of_year
hour(timestamp), hour(time), hour(interval day to second)	bigint	`SELECT hour('2019-10-12 23:10:34.000000000')` Example result: 23

Function	Output data type	Description
millisecond(timestamp), millisecond(time), millisecond(interval day to second)	bigint	`SELECT millisecond('2019-10-12 23:10:34.000000000')` Example result: `0`
minute(timestamp), minute(time), minute(interval day to second)	bigint	`SELECT minute('2019-10-12 23:10:34.000000000')` Example result: `10`
month(timestamp), month(date), month(interval year to month)	bigint	`SELECT month('2019-10-12 23:10:34.000000000')` Example result: `10`
nanosecond(timestamp), nanosecond(time), nanosecond(interval day to second)	bigint	`SELECT nanosecond(current_timestamp)` Example result: `162000000`
quarter(timestamp), quarter(date)	bigint	`SELECT quarter('2019-10-12 23:10:34.000000000')` Example result: `4`
second(timestamp), second(time), second(interval day to second)	bigint	`SELECT second('2019-10-12 23:10:34.000000000')` Example result: `34`

Function	Output data type	Description
week(timestamp), week(date)	bigint	`SELECT week('2019-10-12 23:10:34.000000000')` Example result: 41
week_of_year(timestamp), week_of_year(date)	bigint	Alias for week
year(timestamp), year(date), year(interval year to month)	bigint	`SELECT year('2019-10-12 23:10:34.000000000')` Example result: 2019
year_of_week(timestamp), year_of_week(date)	bigint	`SELECT year_of_week('2019-10-12 23:10:34.000000000')` Example result: 2019
yow(timestamp), yow(date)	bigint	Alias for year_of_week

Aggregate functions

Timestream for LiveAnalytics supports the following aggregate functions.

Function	Output data type	Description
arbitrary(x)	[same as input]	Returns an arbitrary non-null value of x, if one exists. `SELECT arbitrary(t.c) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: 1

Function	Output data type	Description
array_agg(x)	array<[same as input]	Returns an array created from the input x elements. `SELECT array_agg(t.c) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `[ 1,2,3,4 ]`
avg(x)	double	Returns the average (arithmetic mean) of all input values. `SELECT avg(t.c) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `2.5`
bool_and(boolean) every(boolean)	boolean	Returns TRUE if every input value is TRUE, otherwise FALSE. `SELECT bool_and(t.c) FROM (VALUES true, true, false, true) AS t(c)` Example result: `false`

Function	Output data type	Description
bool_or(boolean)	boolean	Returns TRUE if any input value is TRUE, otherwise FALSE. `SELECT bool_or(t.c) FROM (VALUES true, true, false, true) AS t(c)` Example result: `true`
count(*) count(x)	bigint	count(*) returns the number of input rows. count(x) returns the number of non-null input values. `SELECT count(t.c) FROM (VALUES true, true, false, true) AS t(c)` Example result: 4
count_if(x)	bigint	Returns the number of TRUE input values. `SELECT count_if(t.c) FROM (VALUES true, true, false, true) AS t(c)` Example result: 3

Function	Output data type	Description
geometric_mean(x)	double	Returns the geometric mean of all input values. `SELECT geometric_mean(t.c) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `2.213363839400643`
max_by(x, y)	[same as x]	Returns the value of x associated with the maximum value of y over all input values. `SELECT max_by(t.c1, t.c2) FROM (VALUES (('a', 1)), (('b', 2)), (('c', 3)), (('d', 4))) AS t(c1, c2)` Example result: d
max_by(x, y, n)	array<[same as x]>	Returns n values of x associated with the n largest of all input values of y in descending order of y. `SELECT max_by(t.c1, t.c2, 2) FROM (VALUES (('a', 1)), (('b', 2)), (('c', 3)), (('d', 4))) AS t(c1, c2)` Example result: `[ d,c ]`

Function	Output data type	Description
min_by(x, y)	[same as x]	Returns the value of x associated with the minimum value of y over all input values. `SELECT min_by(t.c1, t.c2) FROM (VALUES (('a', 1)), (('b', 2)), (('c', 3)), (('d', 4))) AS t(c1, c2)` Example result: a
min_by(x, y, n)	array<[same as x]>	Returns n values of x associated with the n smallest of all input values of y in ascending order of y. `SELECT min_by(t.c1, t.c2, 2) FROM (VALUES (('a', 1)), (('b', 2)), (('c', 3)), (('d', 4))) AS t(c1, c2)` Example result: `[ a,b ]`
max(x)	[same as input]	Returns the maximum value of all input values. `SELECT max(t.c) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: 4

Function	Output data type	Description
max(x, n)	array<[same as x]>	Returns n largest values of all input values of x. `SELECT max(t.c, 2) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `[ 4,3 ]`
min(x)	[same as input]	Returns the minimum value of all input values. `SELECT min(t.c) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: 1
min(x, n)	array<[same as x]>	Returns n smallest values of all input values of x. `SELECT min(t.c, 2) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `[ 1,2 ]`
sum(x)	[same as input]	Returns the sum of all input values. `SELECT sum(t.c) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `10`

Function	Output data type	Description
bitwise_and_agg(x)	bigint	Returns the bitwise AND of all input values in 2s complement representation. `SELECT bitwise_and_agg(t.c) FROM (VALUES 1, -3) AS t(c)` Example result: 1
bitwise_or_agg(x)	bigint	Returns the bitwise OR of all input values in 2s complement representation. `SELECT bitwise_or_agg(t.c) FROM (VALUES 1, -3) AS t(c)` Example result: -3

Function	Output data type	Description
approx_distinct(x)	bigint	Returns the approxima te number of distinct input values. This function provides an approximation of count(DISTINCT x). Zero is returned if all input values are null. This function should produce a standard error of 2.3%, which is the standard deviation of the (approxim ately normal) error distribut ion over all possible sets. It does not guarantee an upper bound on the error for any specific input set. `SELECT approx_di stinct(t.c) FROM (VALUES 1, 2, 3, 4, 8) AS t(c)` Example result: 5

Function	Output data type	Description
approx_distinct(x, e)	bigint	Returns the approxima te number of distinct input values. This function provides an approximation of count(DISTINCT x). Zero is returned if all input values are null. This function should produce a standard error of no more than e, which is the standard deviation of the (approximately normal) error distribution over all possible sets. It does not guarantee an upper bound on the error for any specific input set. The current implementation of this function requires that e be in the range of [0.004062 5, 0.26000]. `SELECT approx_di stinct(t.c, 0.2) FROM (VALUES 1, 2, 3, 4, 8) AS t(c)` Example result: 5

Function	Output data type	Description
approx_percentile(x, percentage)	[same as x]	Returns the approximate percentile for all input values of x at the given percentage. The value of percentage must be between zero and one and must be constant for all input rows. `SELECT approx_percentile(t.c, 0.4) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: 2
approx_percentile(x, percentages)	array<[same as x]>	Returns the approximate percentile for all input values of x at each of the specified percentages. Each element of the percentages array must be between zero and one, and the array must be constant for all input rows. `SELECT approx_percentile(t.c, ARRAY[0.1, 0.8, 0.8]) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `[ 1,4,4 ]`

Function	Output data type	Description
approx_percentile(x, w, percentage)	[same as x]	Returns the approximate weighed percentile for all input values of x using the per-item weight w at the percentage p. The weight must be an integer value of at least one. It is effectively a replication count for the value x in the percentile set. The value of p must be between zero and one and must be constant for all input rows. `SELECT approx_percentile(t.c, 1, 0.1) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: 1

Function	Output data type	Description
approx_percentile(x, w, percentages)	array<[same as x]>	Returns the approximate weighed percentile for all input values of x using the per-item weight w at each of the given percentages specified in the array. The weight must be an integer value of at least one. It is effectively a replication count for the value x in the percentil e set. Each element of the array must be between zero and one, and the array must be constant for all input rows. `SELECT approx_pe rcentile(t.c, 1, ARRAY[0.1, 0.8, 0.8]) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: `[ 1,4,4 ]`

Function	Output data type	Description
approx_percentile(x, w, percentage, accuracy)	[same as x]	Returns the approximate weighed percentile for all input values of x using the per-item weight w at the percentage p, with a maximum rank error of accuracy. The weight must be an integer value of at least one. It is effectively a replication count for the value x in the percentile set. The value of p must be between zero and one and must be constant for all input rows. The accuracy must be a value greater than zero and less than one, and it must be constant for all input rows. `SELECT approx_percentile(t.c, 1, 0.1, 0.5) FROM (VALUES 1, 2, 3, 4) AS t(c)` Example result: 1
corr(y, x)	double	Returns correlation coefficient of input values. `SELECT corr(t.c1, t.c2) FROM (VALUES ((1, 1)), ((2, 2)), ((3, 3)), ((4, 4))) AS t(c1, c2)` Example result: `1.0`

Function	Output data type	Description
covar_pop(y, x)	double	Returns the population covariance of input values. `SELECT covar_pop(t.c1, t.c2) FROM (VALUES ((1, 1)), ((2, 2)), ((3, 3)), ((4, 4))) AS t(c1, c2)` Example result: `1.25`
covar_samp(y, x)	double	Returns the sample covariance of input values. `SELECT covar_samp(t.c1, t.c2) FROM (VALUES ((1, 1)), ((2, 2)), ((3, 3)), ((4, 4))) AS t(c1, c2)` Example result: `1.6666666666666667`
regr_intercept(y, x)	double	Returns linear regression intercept of input values. y is the dependent value. x is the independent value. `SELECT regr_intercept(t.c1, t.c2) FROM (VALUES ((1, 1)), ((2, 2)), ((3, 3)), ((4, 4))) AS t(c1, c2)` Example result: `0.0`

Function	Output data type	Description
regr_slope(y, x)	double	Returns linear regression slope of input values. y is the dependent value. x is the independent value. `SELECT regr_slope(t.c1, t.c2) FROM (VALUES ((1, 1)), ((2, 2)), ((3, 3)), ((4, 4))) AS t(c1, c2)` Example result: `1.0`
skewness(x)	double	Returns the skewness of all input values. `SELECT skewness(t.c1) FROM (VALUES 1, 2, 3, 4, 8) AS t(c1)` Example result: `0.8978957037987335`
stddev_pop(x)	double	Returns the population standard deviation of all input values. `SELECT stddev_pop(t.c1) FROM (VALUES 1, 2, 3, 4, 8) AS t(c1)` Example result: `2.4166091947189146`

Function	Output data type	Description
stddev_samp(x) stddev(x)	double	Returns the sample standard deviation of all input values. `SELECT stddev_samp(t.c1) FROM (VALUES 1, 2, 3, 4, 8) AS t(c1)` Example result: `2.701851217221259`
var_pop(x)	double	Returns the population variance of all input values. `SELECT var_pop(t.c1) FROM (VALUES 1, 2, 3, 4, 8) AS t(c1)` Example result: `5.840000000000001`
var_samp(x) variance(x)	double	Returns the sample variance of all input values. `SELECT var_samp(t.c1) FROM (VALUES 1, 2, 3, 4, 8) AS t(c1)` Example result: `7.300000000000001`

Window functions

Window functions perform calculations across rows of the query result. They run after the HAVING clause but before the ORDER BY clause. Invoking a window function requires special syntax using the OVER clause to specify the window. A window has three components:

- The partition specification, which separates the input rows into different partitions. This is analogous to how the GROUP BY clause separates rows into different groups for aggregate functions.
- The ordering specification, which determines the order in which input rows will be processed by the window function.
- The window frame, which specifies a sliding window of rows to be processed by the function for a given row. If the frame is not specified, it defaults to RANGE UNBOUNDED PRECEDING, which is the same as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. This frame contains all rows from the start of the partition up to the last peer of the current row.

All Aggregate Functions can be used as window functions by adding the OVER clause. The aggregate function is computed for each row over the rows within the current row's window frame. In addition to aggregate functions, Timestream for LiveAnalytics supports the following ranking and value functions.

Function	Output data type	Description
cume_dist()	bigint	Returns the cumulative distribution of a value in a group of values. The result is the number of rows preceding or peer with the row in the window ordering of the window partition divided by the total number of rows in the window partition. Thus, any tie values in the ordering will evaluate to the same distribution value.
dense_rank()	bigint	Returns the rank of a value in a group of values. This is similar to rank(), except that tie values do not produce gaps in the sequence.

Function	Output data type	Description
ntile(n)	bigint	Divides the rows for each window partition into n buckets ranging from 1 to at most n. Bucket values will differ by at most 1. If the number of rows in the partition does not divide evenly into the number of buckets, then the remainder values are distributed one per bucket, starting with the first bucket.
percent_rank()	double	Returns the percentage ranking of a value in group of values. The result is (r - 1) / (n - 1) where r is the rank() of the row and n is the total number of rows in the window partition.
rank()	bigint	Returns the rank of a value in a group of values. The rank is one plus the number of rows preceding the row that are not peer with the row. Thus, tie values in the ordering will produce gaps in the sequence. The ranking is performed for each window partition.

Function	Output data type	Description
row_number()	bigint	Returns a unique, sequential number for each row, starting with one, according to the ordering of rows within the window partition.
first_value(x)	[same as input]	Returns the first value of the window. This function is scoped to the window frame. The function takes an expression or target as its parameter.
last_value(x)	[same as input]	Returns the last value of the window. This function is scoped to the window frame. The function takes an expression or target as its parameter.
nth_value(x, offset)	[same as input]	Returns the value at the specified offset from beginning the window. Offsets start at 1. The offset can be any scalar expression. If the offset is null or greater than the number of values in the window, null is returned. It is an error for the offset to be zero or negative. The function takes an expression or target as its first parameter .

Function	Output data type	Description
lead(x[, offset[, default_value]])	[same as input]	Returns the value at offset rows after the current row in the window. Offsets start at 0, which is the current row. The offset can be any scalar expression. The default offset is 1. If the offset is null or larger than the window, the default_value is returned, or if it is not specified null is returned. The function takes an expression or target as its first parameter.
lag(x[, offset[, default_value]])	[same as input]	Returns the value at offset rows before the current row in the window Offsets start at 0, which is the current row. The offset can be any scalar expression. The default offset is 1. If the offset is null or larger than the window, the default_value is returned, or if it is not specified null is returned. The function takes an expression or target as its first parameter.

Sample queries

This section includes example use cases of Timestream for LiveAnalytics's query language.

Topics

- Simple queries

- Queries with time series functions
- Queries with aggregate functions

Simple queries

The following gets the 10 most recently added data points for a table.

```
SELECT * FROM <database_name>.<table_name>
ORDER BY time DESC
LIMIT 10
```

The following gets the 5 oldest data points for a specific measure.

```
SELECT * FROM <database_name>.<table_name>
WHERE measure_name = '<measure_name>'
ORDER BY time ASC
LIMIT 5
```

The following works with nanosecond granularity timestamps.

```
SELECT now() AS time_now
, now() - (INTERVAL '12' HOUR) AS twelve_hour_earlier -- Compatibility with ANSI SQL
, now() - 12h AS also_twelve_hour_earlier -- Convenient time interval literals
, ago(12h) AS twelve_hours_ago -- More convenience with time functionality
, bin(now(), 10m) AS time_binned -- Convenient time binning support
, ago(50ns) AS fifty_ns_ago -- Nanosecond support
, now() + (1h + 50ns) AS hour_fifty_ns_future
```

Measure values for multi-measure records are identified by column name. Measure values for single-measure records are identified by `measure_value::<data_type>`, where *<data_type>* is one of `double`, `bigint`, `boolean`, or `varchar` as described in Supported data types. For more information about how measure values are modeled, see Single table vs. multiple tables.

The following retrieves values for a measure called `speed` from multi-measure records with a `measure_name` of `IoTMulti-stats`.

```
SELECT speed FROM <database_name>.<table_name> where measure_name = 'IoTMulti-stats'
```

The following retrieves `double` values from single-measure records with a `measure_name` of `load`.

```
SELECT measure_value::double FROM <database_name>.<table_name> WHERE measure_name =
 'load'
```

Queries with time series functions

Topics

- Example dataset and queries

Example dataset and queries

You can use Timestream for LiveAnalytics to understand and improve the performance and availability of your services and applications. Below is an example table and sample queries run on that table.

The table `ec2_metrics` stores telemetry data, such as CPU utilization and other metrics from EC2 instances. You can view the table below.

Time	region	az	Hostname	measure_name	measure_value::double	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	35.1	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	memory_utilization	55.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	network_bytes_in	null	1,500
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	network_bytes_out	null	6,700

Time	region	az	Hostname	measure_name	measure_value::double	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	cpu_utilization	38.5	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	memory_utilization	58.4	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	network_bytes_in	null	23,000
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	network_bytes_out	null	12,000
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	cpu_utilization	45.0	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	memory_utilization	65.8	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	network_bytes_in	null	15,000
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	network_bytes_out	null	836,000
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	55.2	null

Time	region	az	Hostname	measure_name	measure_value::double	measure_value::bigint
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	memory_utilization	75.0	null
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	network_bytes_in	null	1,245
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	network_bytes_out	null	68,432
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	cpu_utilization	65.6	null
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	memory_utilization	85.3	null
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	network_bytes_in	null	1,245
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	network_bytes_out	null	68,432
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	cpu_utilization	12.1	null
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	memory_utilization	32.0	null

Time	region	az	Hostname	measure_name	measure_value::double	measure_value::bigint
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	network_bytes_in	null	1,400
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	network_bytes_out	null	345
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	15.3	null
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	memory_utilization	35.4	null
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	network_bytes_in	null	23
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	network_bytes_out	null	0
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	cpu_utilization	44.0	null
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	memory_utilization	64.2	null
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	network_bytes_in	null	1,450

Time	region	az	Hostname	measure_name	measure_value::double	measure_value::bigint
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	network_bytes_out	null	200
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	cpu_utilization	66.4	null
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	memory_utilization	86.3	null
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	network_bytes_in	null	300
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	network_bytes_out	null	423

Find the average, p90, p95, and p99 CPU utilization for a specific EC2 host over the past 2 hours:

```
SELECT region, az, hostname, BIN(time, 15s) AS binned_timestamp,
    ROUND(AVG(measure_value::double), 2) AS avg_cpu_utilization,
    ROUND(APPROX_PERCENTILE(measure_value::double, 0.9), 2) AS p90_cpu_utilization,
    ROUND(APPROX_PERCENTILE(measure_value::double, 0.95), 2) AS p95_cpu_utilization,
    ROUND(APPROX_PERCENTILE(measure_value::double, 0.99), 2) AS p99_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
    AND time > ago(2h)
GROUP BY region, hostname, az, BIN(time, 15s)
ORDER BY binned_timestamp ASC
```

Identify EC2 hosts with CPU utilization that is higher by 10 % or more compared to the average CPU utilization of the entire fleet for the past 2 hours:

```
WITH avg_fleet_utilization AS (
    SELECT COUNT(DISTINCT hostname) AS total_host_count, AVG(measure_value::double) AS
 fleet_avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND time > ago(2h)
), avg_per_host_cpu AS (
    SELECT region, az, hostname, AVG(measure_value::double) AS avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND time > ago(2h)
    GROUP BY region, az, hostname
)
SELECT region, az, hostname, avg_cpu_utilization, fleet_avg_cpu_utilization
FROM avg_fleet_utilization, avg_per_host_cpu
WHERE avg_cpu_utilization > 1.1 * fleet_avg_cpu_utilization
ORDER BY avg_cpu_utilization DESC
```

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours:

```
SELECT BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double), 2) AS
 avg_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
    AND time > ago(2h)
GROUP BY hostname, BIN(time, 30s)
ORDER BY binned_timestamp ASC
```

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using linear interpolation:

```
WITH binned_timeseries AS (
    SELECT hostname, BIN(time, 30s) AS binned_timestamp,
 ROUND(AVG(measure_value::double), 2) AS avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND hostname = 'host-Hovjv'
```

```
        AND time > ago(2h)
    GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
    SELECT hostname,
        INTERPOLATE_LINEAR(
            CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
                SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
    FROM binned_timeseries
    GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using interpolation based on the last observation carried forward:

```
WITH binned_timeseries AS (
    SELECT hostname, BIN(time, 30s) AS binned_timestamp,
 ROUND(AVG(measure_value::double), 2) AS avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND hostname = 'host-Hovjv'
        AND time > ago(2h)
    GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
    SELECT hostname,
        INTERPOLATE_LOCF(
            CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
                SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
    FROM binned_timeseries
    GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Queries with aggregate functions

Below is an example IoT scenario example data set to illustrate queries with aggregate functions.

Topics

- Example data
- Example queries

Example data

Timestream enables you to store and analyze IoT sensor data such as the location, fuel consumption, speed, and load capacity of one or more fleets of trucks to enable effective fleet management. Below is the schema and some of the data of a table iot_trucks that stores telemetry such as location, fuel consumption, speed, and load capacity of trucks.

Time	truck_id	Make	Model	Fleet	fuel_capacity	load_capacity	measure_name	measure_value::double	measure_value::varchar
2019-12-4 19:00:00.000000000	1234567	GMC	Astro	Alpha	100	500	fuel_reading	65.2	null
2019-12-4 19:00:00.000000000	1234567	GMC	Astro	Alpha	100	500	load	400.0	null
2019-12-4 19:00:00.000000000	1234567	GMC	Astro	Alpha	100	500	speed	90.2	null
2019-12-4 19:00:00.000000000	1234567	GMC	Astro	Alpha	100	500	location	null	47.6062 N, 122.3321 W
2019-12-4 19:00:00	1234567	Kenworth	W900	Alpha	150	1000	fuel_reading	10.1	null

Time	truck_id	Make	Model	Fleet	fuel_capacity	load_capacity	measure_name	measure_value::double	measure_value::varchar
0000000									
2019-12-4 19:00:00.000000	1234567	Kenworth	W900	Alpha	150	1000	load	950.3	null
2019-12-4 19:00:00.000000	1234567	Kenworth	W900	Alpha	150	1000	speed	50.8	null
2019-12-4 19:00:00.000000	1234567	Kenworth	W900	Alpha	150	1000	location	null	40.7128 degrees N, 74.0060 degrees W

Example queries

Get a list of all the sensor attributes and values being monitored for each truck in the fleet.

```
SELECT
    truck_id,
    fleet,
    fuel_capacity,
    model,
    load_capacity,
    make,
    measure_name
FROM "sampleDB".IoT
GROUP BY truck_id, fleet, fuel_capacity, model, load_capacity, make, measure_name
```

Get the most recent fuel reading of each truck in the fleet in the past 24 hours.

```
WITH latest_recorded_time AS (
    SELECT
        truck_id,
        max(time) as latest_time
    FROM "sampleDB".IoT
    WHERE measure_name = 'fuel-reading'
    AND time >= ago(24h)
    GROUP BY truck_id
)
SELECT
    b.truck_id,
    b.fleet,
    b.make,
    b.model,
    b.time,
    b.measure_value::double as last_reported_fuel_reading
FROM
latest_recorded_time a INNER JOIN "sampleDB".IoT b
ON a.truck_id = b.truck_id AND b.time = a.latest_time
WHERE b.measure_name = 'fuel-reading'
AND b.time > ago(24h)
ORDER BY b.truck_id
```

Identify trucks that have been running on low fuel(less than 10 %) in the past 48 hours:

```
WITH low_fuel_trucks AS (
    SELECT time, truck_id, fleet, make, model, (measure_value::double/
cast(fuel_capacity as double)*100) AS fuel_pct
    FROM "sampleDB".IoT
    WHERE time >= ago(48h)
    AND (measure_value::double/cast(fuel_capacity as double)*100) < 10
    AND measure_name = 'fuel-reading'
),
other_trucks AS (
SELECT time, truck_id, (measure_value::double/cast(fuel_capacity as double)*100) as
 remaining_fuel
    FROM "sampleDB".IoT
    WHERE time >= ago(48h)
    AND truck_id IN (SELECT truck_id FROM low_fuel_trucks)
    AND (measure_value::double/cast(fuel_capacity as double)*100) >= 10
    AND measure_name = 'fuel-reading'
),
trucks_that_refuelled AS (
```

```
    SELECT a.truck_id
    FROM low_fuel_trucks a JOIN other_trucks b
    ON a.truck_id = b.truck_id AND b.time >= a.time
)
SELECT DISTINCT truck_id, fleet, make, model, fuel_pct
FROM low_fuel_trucks
WHERE truck_id NOT IN (
    SELECT truck_id FROM trucks_that_refuelled
)
```

Find the average load and max speed for each truck for the past week:

```
SELECT
    bin(time, 1d) as binned_time,
    fleet,
    truck_id,
    make,
    model,
    AVG(
        CASE WHEN measure_name = 'load' THEN measure_value::double ELSE NULL END
    ) AS avg_load_tons,
    MAX(
        CASE WHEN measure_name = 'speed' THEN measure_value::double ELSE NULL END
    ) AS max_speed_mph
FROM "sampleDB".IoT
WHERE time >= ago(7d)
AND measure_name IN ('load', 'speed')
GROUP BY fleet, truck_id, make, model, bin(time, 1d)
ORDER BY truck_id
```

Get the load efficiency for each truck for the past week:

```
WITH average_load_per_truck AS (
    SELECT
        truck_id,
        avg(measure_value::double)  AS avg_load
    FROM "sampleDB".IoT
    WHERE measure_name = 'load'
    AND time >= ago(7d)
    GROUP BY truck_id, fleet, load_capacity, make, model
),
truck_load_efficiency AS (
    SELECT
```

```
        a.truck_id,
        fleet,
        load_capacity,
        make,
        model,
        avg_load,
        measure_value::double,
        time,
        (measure_value::double*100)/avg_load as load_efficiency -- ,
 approx_percentile(avg_load_pct, DOUBLE '0.9')
    FROM "sampleDB".IoT a JOIN average_load_per_truck b
    ON a.truck_id = b.truck_id
    WHERE a.measure_name = 'load'
)
SELECT
    truck_id,
    time,
    load_efficiency
FROM truck_load_efficiency
ORDER BY truck_id, time
```

API reference

For the Amazon Timestream for LiveAnalytics API Reference, see the Amazon Timestream for LiveAnalytics API Reference.

For the Amazon Timestream for InfluxDB API Reference, see the Amazon Timestream for InfluxDB API Reference.

Document history

Change	Description	Date
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `CreateDbBackup`, `GetDbBackup`, `ListDbBackups`, `DeleteDbBackup`, and `RestoreFromDbBackup` actions to the `AmazonTim`	August 4, 2026

	`estreamInfluxDBFullAccessWithoutMarketplaceAccess` managed policy. These actions support backup and restore operations on Amazon Timestream for InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	
`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added the `CreateDbBackup`, `GetDbBackup`, `ListDbBackups`, `DeleteDbBackup`, and `RestoreFromDbBackup` actions to the `AmazonTimestreamInfluxDBFullAccess` managed policy. These actions support backup and restore operations on Amazon Timestream for InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	August 4, 2026

AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the RebootDbInstance and RebootDbCluster actions to the existing AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess managed policy for rebooting Amazon Timestream InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	December 17, 2025
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the RebootDbInstance and RebootDbCluster actions to the existing AmazonTimestreamInfluxDBFullAccess managed policy for rebooting Amazon Timestream InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	December 17, 2025

AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `ec2:DescribeVpcEndpoints` action to the existing `AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` managed policy for describing the VPC endpoints. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	November 13, 2025
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `ec2:DescribeVpcEndpoints` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy for describing the VPC endpoints. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	November 13, 2025

`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added Influx Enterprise marketplace product ID to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy to support subscription to enterprise marketplace offerings. See AmazonTimestreamInfluxDBFullAccess.	October 17, 2025
`AmazonTimestreamConsoleFullAccess` – Update to an existing policy	Timestream for LiveAnalytics has added the AWS Marketplace permissions to the `AmazonTimestreamConsoleFullAccess` managed policy to access marketplace resources and create agreements for InfluxDB Cluster with Read Replicas creation.	August 20, 2025
`AmazonTimestreamConsoleFullAccess` – Update to an existing policy	Timestream for InfluxDB has added the AWS Marketplace permissions to the `AmazonTimestreamConsoleFullAccess` managed policy to access marketplace resources and create agreements for InfluxDB Cluster with Read Replicas creation.	August 20, 2025

AmazonTimestreamConsoleFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `pricing:GetProducts` permission to the `AmazonTimestreamConsoleFullAccess` managed policy to provide pricing estimations for InfluxDB resource configurations during creation.	June 10, 2025
Amazon Timestream for LiveAnalytics will no longer be open to new customers starting June 20, 2025.	For similar capabilities to Amazon Timestream for LiveAnalytics, consider Amazon Timestream for InfluxDB 3 Enterprise for large-scale time-series analytics with unlimited cardinality, native SQL support, and multi-node scaling. For operational workloads requiring single-digit millisecond query response times, consider Amazon Timestream for InfluxDB 2. Learn more here.	May 20, 2025
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – New policy	This policy grants administrative permissions that allow full access to all Timestream for InfluxDB resources, excluding any marketplace-related actions. For more information see AWS managed policies for Amazon Timestream for InfluxDB.	April 16, 2025

`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	April 16, 2025
`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added access to create, update, delete, and list Amazon Timestream InfluxDB clusters to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	February 17, 2025
Documentation-only update	Updated the Quotas topic to segregate the default quotas and system limits.	October 22, 2024
Amazon Timestream now supports query insights	Timestream now includes support for the query insights feature that helps you optimize your queries, improve their performance, and reduce costs.	October 22, 2024

Amazon Timestream for InfluxDB update to an existing policy.	Amazon Timestream for InfluxDB has added the `ec2:DescribeRouteTables` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy for describing your route tables. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	October 8, 2024
`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added the `ec2:DescribeRouteTables` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy. This action is used for describing your route tables. See AmazonTimestreamInfluxDBFullAccess.	September 12, 2024
`AmazonTimestreamReadOnlyAccess` – Update to an existing policy	Timestream for LiveAnalytics has added the `DescribeAccountSettings` permission to the `AmazonTimestreamReadOnlyAccess` managed policy for describing AWS account settings.	June 3, 2024

Amazon Timestream for LiveAnalytics now supports Timestream Compute Units (TCUs)	Amazon Timestream for LiveAnalytics now includes support for Timestream Compute Units (TCUs) to measure the compute capacity allocated for your query needs.	April 29, 2024
New policies added	Amazon Timestream for InfluxDB added two new policies: One that allows the service to manage network interfaces and security groups in your account. For more information, see AmazonTimestreamInfluxDBServiceRolePolicy. Another that provide full administrative access to create, update, delete and list Amazon Timestream InfluxDB instances and create and list parameter groups. For more information, see AmazonTimestreamInfluxDBFullAccess.	March 14, 2024
Amazon Timestream for InfluxDB is now generally available.	This documentation covers the initial release of Amazon Timestream for InfluxDB.	March 14, 2024

Amazon Timestream for LiveAnalytics Query events are available in AWS CloudTrail	Amazon Timestream for LiveAnalytics now publishes Query API data events to AWS CloudTrail. Customers can audit all Query API requests made in their AWS accounts, and see information such as which IAM User/Role made the request, when the request was made, which databases and tables were queried, and the request's Query ID.	September 12, 2023
Amazon Timestream for LiveAnalytics UNLOAD	Amazon Timestream for LiveAnalytics now supports UNLOAD to export query results to S3.	May 12, 2023
Amazon Timestream for LiveAnalytics update to an existing policy.	Batch load permissions added to a managed policy.	February 24, 2023
Amazon Timestream for LiveAnalytics batch load.	Amazon Timestream for LiveAnalytics now supports batch load functionality.	February 24, 2023
Amazon Timestream for LiveAnalytics now supports AWS Backup.	Amazon Timestream for LiveAnalytics now supports AWS Backup.	December 14, 2022
Amazon Timestream for LiveAnalytics updates to AWS managed policies	New information about AWS managed policies and Amazon Timestream for LiveAnalytics, including updates to existing managed policies.	November 29, 2021

Amazon Timestream for LiveAnalytics supports scheduled queries	Amazon Timestream for LiveAnalytics now supports running a query on your behalf, based on a schedule.	November 29, 2021
Amazon Timestream for LiveAnalytics supports magnetic store.	Amazon Timestream for LiveAnalytics now supports using magnetic storage for your table writes.	November 29, 2021
Amazon Timestream for LiveAnalytics multi-measure records.	Amazon Timestream for LiveAnalytics now supports a more compact format for storing your time-series data.	November 29, 2021
Amazon Timestream for LiveAnalytics updates to AWS managed policies	New information about AWS managed policies and Amazon Timestream for LiveAnalytics, including updates to existing managed policies.	May 24, 2021
Amazon Timestream for LiveAnalytics is now available in the Europe (Frankfurt) region.	Amazon Timestream for LiveAnalytics is now generally available in the Europe (Frankfurt) region (`eu-central-1`).	April 23, 2021
Amazon Timestream for LiveAnalytics now supports VPC endpoints (AWS PrivateLink).	Amazon Timestream for LiveAnalytics now supports the use of VPC endpoints (AWS PrivateLink).	March 23, 2021
Amazon Timestream now supports cross table queries.	You can use Amazon Timestream for LiveAnalytics to run cross table queries.	February 10, 2021

Amazon Timestream for LiveAnalytics now supports enhanced query execution statistics.	Amazon Timestream for LiveAnalytics now supports enhanced query execution statistics, such as amount of data scanned.	February 10, 2021
Amazon Timestream for LiveAnalytics now supports advanced time series functions.	You can use Amazon Timestream for LiveAnalytics to run SQL queries with advanced time series functions, such as derivatives, integrals, and correlations.	February 10, 2021
Amazon Timestream for LiveAnalytics is now HIPAA, ISO, and PCI compliant.	You can now use Amazon Timestream for LiveAnalytics for workloads that require HIPAA, ISO, and PCI-compliant infrastructure.	January 27, 2021
Amazon Timestream for LiveAnalytics now supports open-source Telegraf and Grafana.	You can now use Telegraf, the open-source, plugin-driven server agent for collecting and reporting metrics, and Grafana, the open-source analytics and monitoring platform for databases, with Amazon Timestream for LiveAnalytics.	November 25, 2020
Amazon Timestream for LiveAnalytics is now generally available.	This documentation covers the initial release of Amazon Timestream for LiveAnalytics.	September 30, 2020

What is Timestream for InfluxDB?

Amazon Timestream for InfluxDB is a managed time series database engine that makes it easy for application developers and DevOps teams to run InfluxDB databases on AWS for real-time time series applications using open-source APIs. With Amazon Timestream for InfluxDB, it is easy to set up, operate, and scale time series workloads that can answer queries with single-digit millisecond query response time.

Amazon Timestream for InfluxDB gives you access to the capabilities of the familiar open source version of InfluxDB on its 2.x branch. This means that the code, applications, and tools you already use today with your existing InfluxDB open-source databases should work seamlessly with Amazon Timestream for InfluxDB. Amazon Timestream for InfluxDB can automatically back up your database and keep your database software up to date with the latest version. In addition, Amazon Timestream for InfluxDB makes it easy to use replication to enhance database availability, and improve data durability. As with all AWS services, there are no upfront investments required, and you pay only for the resources you use.

DB instances

A DB instance is an isolated database environment running in the cloud. It is the basic building block of Amazon Timestream for InfluxDB. A DB instance can contain multiple user-created databases (or organizations and buckets for the case of InfluxDb 2.x databases), and can be accessed using the same client tools and applications you might use to access a standalone self-managed InfluxDB instance. DB instances are simple to create and modify with the AWS command line tools, Amazon Timestream InfluxDB API operations, or the AWS Management Console.

> **Note**
>
> Amazon Timestream for InfluxDB supports access to databases using the Influx API operations and Influx UI. Amazon Timestream for InfluxDB does not allow direct host access.

You can have up to 40 Amazon Timestream for InfluxDB instances.

Each DB instance has a DB instance id. This service generated name uniquely identifies the DB instance when interacting with the Amazon Timestream for InfluxDB API and AWS CLI commands. The DB instance id is unique for that customer in an AWS Region.

The DB instance id forms part of the DNS hostname allocated to your instance by Timestream for InfluxDB. For example, if you specify influxdb1 as the DB instance name and the service generates an instance id *c5vasdqn0b* then Timestream will automatically allocate a DNS endpoint for your instance. An example endpoint is `c5vasdqn0b-3ksj4dla5nfjhi.timestream-influxdb.us-east-1.on.aws`, where `c5vasdqn0b` is your instance id. All instances created before 12/09/2024 will maintain the old structure with an endpoint similar to: `influxdb1-3ksj4dla5nfjhi.us-east-1.timestream-influxdb.amazonaws.com` where `influxdb1` is your instance name.

In the example endpoint `c5vasdqn0b-3ksj4dla5nfjhi.timestream-influxdb.us-east-1.on.aws`, the string `3ksj4dla5nfjhi` is a unique account identifier generated by AWS. The identifier `3ksj4dla5nfjhi` in the example doesn't change for the specified account in a certain Region. Therefore, all your DB instances created by this account share the same fixed identifier in the Region. Consider the following features of the fixed identifier:

- Currently Timestream for InfluxDB does not support DB instance renaming.
- For all instances created after 12/09/2024, if you delete and re-create your DB instance with the same DB instance name, the endpoint will change since a new instance id will be assigned to the instance. Instance created before tthe aforementioned date will be assigned the same endpoint based on instance name.
- If you use the same account to create a DB instance in a different Region, the internally generated identifier is different because the Region is different, as in `zxlasoonhvd.4a3j5du5ks7md2.timestream-influxdb.us-east-1.on.aws`.

Each DB instance supports only one Timestream for InfluxDB database engine.

When creating a DB instance, InfluxDB requires that an organization name be specified. A DB instance can host multiple organizations and multiple buckets associated to each organization.

Amazon Timestream for InfluxDB allows you to create a master user account and password for your DB instance as part of the creation process. This master user has permissions to create organizations, buckets, and to perform read, write, delete and upsert operations on your data. You will also be able to access the InfluxUI and retrieve you operator token on. your first log in. From there you will be able to manage all your access tokens as well. You must set the master user password when you create a DB instance, but you can change it at any time using the Influx API, Influx CLI, or the InfluxUI.

DB instance classes

The DB instance class determines the computation and memory capacity of an Amazon Timestream for InfluxDB DB instance. The DB instance class that you need depends on your processing power and memory requirements.

A DB instance class consists of both the DB instance class type and the size. For example, `db.influx` is a memory-optimized DB instance class type suitable for the high performance memory requirements related to running InfluxDb workloads. Within the `db.influx` instance class type, `db.influx.2xlarge` is a DB instance class. The size of this class is *2xlarge*.

For more information about instance class pricing, see Amazon Timestream for InfluxDB pricing.

DB instance class types

Amazon Timestream for InfluxDB supports DB instance classes for the following use case optimized for InfluxDB use cases.

- **`db.influx`**—These instance classes are ideal for running memory-intensive workloads in open-source InfluxDB databases

Hardware specifications for DB instance classes

The following terminology describes the hardware specifications for DB instance classes:

- **vCPU**

  The number of virtual central processing units (CPUs). A virtual CPU is a unit of capacity that you can use to compare DB instance classes.

- **Memory (GiB)**

  The RAM, in gibibytes, allocated to the DB instance. There is often a consistent ratio between memory and vCPU. As an example, take the db.influx instance class, which has a memory to vCPU ratio similar to the EC2 r7g instance class.

- **Influx-Optimized**

The DB instance uses an optimized configuration stack and provides additional, dedicated capacity for I/O. This optimization provides the best performance by minimizing contention between I/O and other traffic from your instance.

- **Network bandwidth**

  The network speed relative to other DB instance classes. In the following table, you can find hardware details about the Amazon Timestream for InfluxDB instance classes.

Instances Class	vCPU	Memory (GiB)	Storage Type	Network bandwidth (Gbps)
db.influx .medium	1	8	Influx IOPS Included	10
db.influx.large	2	16	Influx IOPS Included	10
db.influx.xlarge	4	32	Influx IOPS Included	10
db.influx.2xlarge	8	64	Influx IOPS Included	10
db.influx.4xlarge	16	128	Influx IOPS Included	10
db.influx.8xlarge	32	256	Influx IOPS Included	12
db.influx .12xlarge	48	384	Influx IOPS Included	20
db.influx .16xlarge	64	512	Influx IOPS Included	25

Instances Class	vCPU	Memory (GiB)	Storage Type	Network bandwidth (Gbps)
db.influx .24xlarge	96	768	Influx IOPS Included	40

InfluxDB instance storage

DB instances for Amazon Timestream for InfluxDB use Influx IOPS Included volumes for databases and log storage.

In some cases, your database workload might not be able to achieve 100 percent of the IOPS that you have provisioned. For more information, see Factors that affect storage performance. For more information about Timestream for InfluxDB storage pricing, see Amazon Timestream pricing.

Amazon Timestream for InfluxDB storage types

Amazon Timestream for InfluxDB provides support for one storage type, Influx IOPS Included. You can create Timestream for InfluxDB instances with up to 16 tebibytes (TiB) of storage.

Here is a brief description of the available storage type:

- **Influx IO Included storage:** Storage performance is the combination of I/O operations per second (IOPS) and how fast the storage volume can perform reads and writes (storage throughput). On Influx IOPS Included storage volumes, Amazon Timestream for InfluxDB provides 3 storage tiers that come pre configured with optimal IOPS and throughput required for different types of workloads.

InfluxDB instance sizing

The optimal configuration of a Timestream for InfluxDB instance depends on various factors, including ingestion rate, batch sizes, time series cardinality, concurrent queries, and query types. To provide sizing recommendations, let's consider an exemplary workload with the following characteristics:

- Data is collected and written by a fleet of Telegraf agents gathering System, CPU, Memory, Disk, IO, and etc. from a data center.

Each write request contains 5000 lines.

- The queries executed on the system are categorized as “moderate complexity” queries, exhibiting the following characteristics:
  - They have multiple functions and one or two regular expressions
  - They may include group by clauses or sample a time range of multiple weeks.
  - They typically takes a few hundred milliseconds to a couple of thousand milliseconds to execute.
  - The CPU favors query performance primarily.

Max # of series	Writes (lines per second)	Reads (Queries per second)	Instance class	Storage Type
<100K	~50,000	<10	db.influx.large	Influx IO Included 3K
<1MM	~150,000	<25	db.influx.2xlarge	Influx IO Included 3K
~1MM	~200,000	~25	db.influx.4xlarge	Influx IO Included 3K
<5MM	~250,000	~35	db.influx.4xlarge	Influx IO Included 12K
<10MM	~500,000	~50	db.influx.8xlarge	Influx IO Included 12K
~10MM	<750,000	<100	db.influx .12xlarge	Influx IO Included 12K

AWS Regions and Availability Zones

Amazon cloud computing resources are hosted in multiple locations world-wide. These locations are composed of AWS Regions and . Each AWS Region is a separate geographic area. Each AWS Region has multiple, isolated locations known as Availability Zones.

> **Note**
>
> For information about finding the for an AWS Region, see Regions and Zones in the *Amazon EC2 User Guide*.

Amazon Timestream for InfluxDB enables you to place resources, such as DB instances, and data in multiple locations.

Amazon operates state-of-the-art, highly-available data centers. Although rare, failures can occur that affect the availability of DB instances that are in the same location. If you host all your DB instances in one location that is affected by such a failure, none of your DB instances will be available.

It is important to remember that each AWS Region is completely independent. Any Amazon Timestream for InfluxDB activity you initiate (for example, creating database instances or listing available database instances) runs only in your current default AWS Region. The default AWS Region can be changed in the console, or by setting the `AWS_DEFAULT_REGION` environment variable. Or it can be overridden by using the `--region` parameter with the AWS Command Line Interface (AWS CLI). For more information, see Configuring the AWS Command Line Interface, specifically the sections about environment variables and command line options.

To create or work with an Amazon Timestream for InfluxDB DB instance in a specific AWS Region, use the corresponding regional service endpoint.

AWS Region availability

The following table shows the AWS Regions where Amazon Timestream for InfluxDB is currently available and the endpoint for each Region.

AWS Region name	Region	Endpoint	Protocol
US East (N. Virginia)	us-east-1	timestream-influxdb.us-east-1.amazonaws.com	HTTPS
US East (Ohio)	us-east-2	timestream-influxdb.us-east-2.amazonaws.com	HTTPS
US West (Oregon)	us-west-2	timestream-influxdb.us-west-2.amazonaws.com	HTTPS
Asia Pacific (Mumbai)	ap-south-1	timestream-influxdb.ap-south-1.amazonaws.com	HTTPS
Asia Pacific (Singapore)	ap-southeast-1	timestream-influxdb.ap-southeast-1.amazonaws.com	HTTPS
Asia Pacific (Sydney)	ap-southeast-2	timestream-influxdb.ap-southeast-2.amazonaws.com	HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	timestream-influxdb.ap-northeast-1.amazonaws.com	HTTPS

AWS Region name	Region	Endpoint	Protocol
Europe (Frankfurt)	eu-central-1	timestream-influxdb.eu-central-1.amazonaws.com	HTTPS
Europe (Ireland)	eu-west-1	timestream-influxdb.eu-west-1.amazonaws.com	HTTPS
Europe (Stockholm)	eu-north-1	timestream-influxdb.eu-north-1.amazonaws.com	HTTPS
Canada (Central)	ca-central-1	timestream-influxdb.ca-central-1.amazonaws.com	HTTPS
Europe (London)	eu-west-2	timestream-influxdb.eu-west-2.amazonaws.com	HTTPS
Europe (Paris)	eu-west-3	timestream-influxdb.eu-west-3.amazonaws.com	HTTPS
Asia Pacific (Jakarta)	ap-southeast-3	timestream-influxdb.ap-southeast-3.amazonaws.com	HTTPS
Europe (Milan)	eu-south-1	timestream-influxdb.eu-south-1.amazonaws.com	HTTPS
Europe (Spain)	eu-south-2	timestream-influxdb.eu-south-2.amazonaws.com	HTTPS

AWS Region name	Region	Endpoint	Protocol
Middle East (UAE)	me-central-1	timestream-influxdb.me-central-1.amazonaws.com	HTTPS
China (Beijing)	cn-north-1	timestream-influxdb.cn-north-1.on.amazonwebservices.com.cn	HTTPS
China (Ningxia)	cn-northwest-1	timestream-influxdb.cn-northwest-1.on.amazonwebservices.com.cn	HTTPS

For more information on AWS Regions where Amazon Timestream for InfluxDB is currently available and the endpoint for each Region, see Amazon Timestream endpoints and quotas.

AWS Regions design

Each AWS Region is designed to be isolated from the other AWS Regions. This design achieves the greatest possible fault tolerance and stability.

When you view your resources, you see only the resources that are tied to the AWS Region that you specified. This is because AWS Regions are isolated from each other, and we don't automatically replicate resources across AWS Regions.

AWS Availability Zones

When you create a DB instance, Amazon Timestream for InfluxDB choose one for you randomly based on your subnet configuration. An *Availability Zone* is represented by an AWS Region code followed by a letter identifier (for example, `us-east-1a`).

Use the `describe-availability`-zones Amazon EC2 command as follows to describe the within the specified Region that are enabled for your account.

```
aws ec2 describe-availability-zones --region region-name
```

For example, to describe the within the *US East (N. Virginia) Region (us-east-1)* that are enabled for your account, run the following command:

```
aws ec2 describe-availability-zones --region us-east-1
```

You can't choose the for the primary and secondary DB instances in a *Multi-AZ DB deployment*. Amazon Timestream for InfluxDB chooses them for you randomly. For more information about Multi-AZ deployments, see Configuring and managing a multi-AZ deployment.

DB Instance billing for Amazon Timestream for InfluxDB

Amazon Timestream for InfluxDB instances are billed based on the following components:

- **DB instance hours (per hour)** — Based on the DB instance class of the DB instance, for example, db.influx.large. Pricing is listed on a per-hour basis, but bills are calculated down to the second and show times in decimal form. Amazon Timestream for InfluxDB usage is billed in 1-second increments, with a minimum of 10 minutes. For more information, see DB instance classesDB instance classes.
- **Storage (per GiB per month)** — Storage capacity that you have provisioned to your DB instance. For more information, see InfluxDB instance storage.
- **Data transfer (per GB)** — Data transfer in and out of your DB instance from or to the internet and other AWS Regions.

For Amazon Timestream for InfluxDB pricing information, see the Amazon Timestream for InfluxDB pricing page.

Setting up Amazon Timestream for InfluxDB

Before you use Amazon Timestream for InfluxDB for the first time, complete the following tasks:

If you already have an AWS account, know your Amazon Timestream for InfluxDB requirements, and prefer to use the defaults for IAM and Amazon VPC Getting started with Timestream for InfluxDB.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

- Go to the AWS sign in page.
- Choose **Create a new account**and the follow the instructions.

> **Note**
>
> Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an AWS account root user is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing *My Account*.

User management

Create an administrative user

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

Sign in to the AWS Management Console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password. For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*

Turn on multi-factor authentication (MFA) for your root user. For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Grant programmatic access

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options:

Which user needs programmatic access?	To	By
Workforce identity(Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. For the AWS CLI, see Configuring IAM Identity Center authentication with the AWS CLI in the *AWS Command Line Interface User Guide*. For AWS SDKs, tools, and AWS APIs, see Using IAM Identity Center to authenticate AWS SDK and tools in the *AWS SDKs and Tools Reference Guide*.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, SDKs, and APIs.	Following the instructions in Use temporary credentials with AWS resources in the *AWS Identity and Access Management User Guide*.
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, SDKs, and APIs.	Following the instructions for the interface that you want to use. For the AWS CLI, see Authenticating using IAM user credentials for the AWS CLI

Which user needs programmatic access?	To	By
		in the *AWS Command Line Interface User Guide*. For AWS SDKs and tools, see Using long-term credentials to authenticate AWS SDKs and tools in the *AWS SDKs and Tools Reference Guide*. For AWS APIs, see Managing access keys for IAM users in the *AWS Identity and Access Management User Guide*.

Determine requirements

The basic building block of Amazon Timestream for InfluxDB is the DB instance. In a DB instance, you create your buckets. A DB instance provides a network address called an endpoint. Your applications use this endpoint to connect to your DB instance. You will also access your InfluxUI using this same endpoint from your browser. When you create a DB instance, you specify details like storage, memory, database engine and version, network configuration, and security. You control network access to a DB instance through a security group.

Before you create a DB instance and a security group, you must know your DB instance and network needs. Here are some important things to consider:

- **Resource requirements** — What are the memory and processor requirements for your application or service? You use these settings to help you determine what DB instance class to use. For specifications about DB instance classes, see DB instance classes.
- **VPC and security group** — Your DB instance will most likely be in a *virtual private cloud (VPC)*. To connect to your DB instance, you need to set up security group rules. These rules are set up differently depending on what kind of VPC you use and how you use it. For example, you can use: a default VPC or a user-defined VPC.

  The following list describes the rules for each VPC option:

- **Default VPC** — If your AWS account has a default VPC in the current AWS Region, that VPC is configured to support DB instances. If you specify the default VPC when you create the DB instance, make sure to create a *VPC security group* that authorizes connections from the application or service to the Amazon Timestream for InfluxDB DB instance. Use the **Security Group** option on the VPC console or the AWS CLI to create VPC security groups. For more information, see Step 3: Create a VPC security group.

- **User-defined VPC** — If you want to specify a user-defined VPC when you create a DB instance, be aware of the following:
  - Make sure to create a *VPC security group* that authorizes connections from the application or service to the Amazon Timestream for InfluxDB DB instance. Use the **Security Group** option on the VPC console or the AWS CLI to create VPC security groups. For information, see Step 3: Create a VPC security group.
  - The VPC must meet certain requirements in order to host DB instances, such as having at least two subnets, each in a separate Availability Zone. For information, see Amazon VPC and Amazon Timestream for InfluxDB.

- **High availability** — Do you need failover support? On Amazon Timestream for InfluxDB, a Multi-AZ deployment creates a primary DB instance and a secondary standby DB instance in another Availability Zone for failover support. We recommend Multi-AZ deployments for production workloads to maintain high availability. For development and test purposes, you can use a deployment that isn't Multi-AZ. For more information, see Multi-AZ DB instance deployments.

- **IAM policies** — Does your AWS account have policies that grant the permissions needed to perform Amazon Timestream for InfluxDB operations? If you are connecting to AWS using IAM credentials, your IAM account must have IAM policies that grant the permissions required to perform Amazon Timestream for InfluxDB control plane operations. For more information, see Identity and Access Management for Amazon Timestream for InfluxDB.

- **Open ports** — What TCP/IP port does your database listen on? The firewalls at some companies might block connections to the default port for your database engine. The default for Timestream for InfluxDB is 8086.

- **AWS Region** — What AWS Region do you want your database in? Having your database in close proximity to your application or web service can reduce network latency. For more information, see AWS Regions and Availability Zones.

- **DB disk subsystem** — What are your storage requirements? Amazon Timestream for InfluxDB provides provides three configurations for it Influx IOPS Included storage type::
  - Influx Io Included 3k IOPS (SSD)

- Influx Io Included 12k IOPS (SSD)
- Influx Io Included 16k IOPS (SSD)

For more information on Amazon Timestream for InfluxDB storage, see Amazon Timestream for InfluxDB DB instance storage. When you have the information you need to create the security group and the DB instance, continue to the next step.

Provide access to your DB instance in your VPC by creating a security group

VPC security groups provide access to DB instances in a VPC. They act as a firewall for the associated DB instance, controlling both inbound and outbound traffic at the DB instance level. DB instances are created by default with a firewall and a default security group that protect the DB instance.

Before you can connect to your DB instance, you must add rules to a security group that enable you to connect. Use your network and configuration information to create rules to allow access to your DB instance.

For example, suppose that you have an application that accesses a database on your DB instance in a VPC. In this case, you must add a custom TCP rule that specifies the port range and IP addresses that your application uses to access the database. If you have an application on an Amazon EC2 instance, you can use the security group that you set up for the Amazon EC2 instance.

Creating a security group for VPC access

To create a VPC security group, sign in to the AWS Management Console and choose VPC.

> **Note**
>
> Make sure you are in the VPC console, not the Amazon Timesteam for InfluxDB console.

- In the upper-right corner of the AWS Management Console, choose the **AWS Region** where you want to create your VPC security group and DB instance. In the list of Amazon VPC resources for that AWS Region, you should see at least one VPC and several subnets. If you don't, you *don't have a default VPC in that AWS Region.*.

- In the navigation pane, choose **Security Groups**.
- Choose **Create security group**.
- Inn the **Basic details** section of the security group page, enter the **Security group name** and **Description**. For **VPC**, choose the VPC thatyou want to create your DB instance in.
- In **Inbound rules**, choose **Add rule**.
  - For **Type**, choose **Custom TCP**.
  - For **Source**, choose a **Security group name** or enter the **IP address range (CIDR value)** from where you access the DB instance. If you choose **My IP**, this allows access to the DB instance from the IP address detected in your browser.

  For Source, choose a security group name or type the IP address range (CIDR value) from where you access the DB instance. If you choose My IP, this allows access to the DB instance from the IP address detected in your browser.
- (Optional) In **Outbound rules**, add rules for outbound traffic. By default, all outbound traffic is allowed.
- Choose **Create security group**.

You can use this *VPC security group* as the security group for your DB instance when you create it.

> **Note**
>
> If you use a *default VPC*, a default subnet group spanning all of the VPC's subnets is created for you. When you create a DB instance, you can choose the *default eiifccntf VPC* and choose *default* for DB Subnet Group.

After you have completed the setup requirements, you can create a DB instance using your requirements and security group. To do so, follow the instructions in Creating a DB instance.

Getting started with Timestream for InfluxDB

In the following examples, you can find out how to create and connect to a DB instance using Amazon Timestream for InfluxDB Service.

> **Note**
>
> Before you can create or connect to a DB instance, make sure to complete the tasks in Setting up Amazon Timestream for InfluxDB.

Topics

- Creating and connecting to a Timestream for InfluxDB instance
- Creating a new operator token for your InfluxDB instance

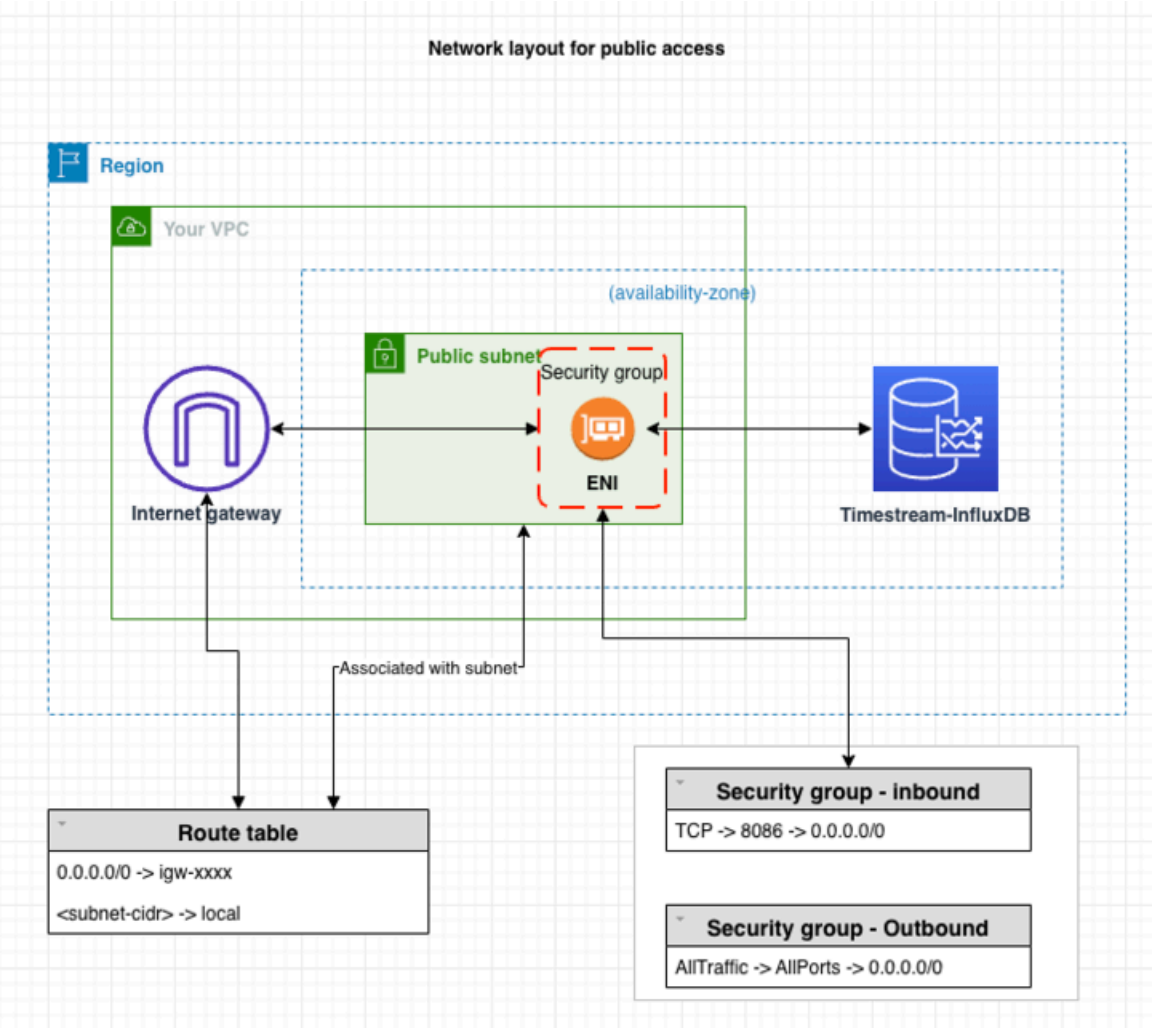
Creating and connecting to a Timestream for InfluxDB instance

This tutorial creates an Amazon EC2 instance and an Amazon Timestream for InfluxDB DB instance. The tutorial shows you how to write data to the DB instance from the EC2 instance using the Telegraf client. As a best practice, this tutorial creates a private DB instance in a virtual private cloud (VPC). In most cases, other resources in the same VPC, such as EC2 instances, can access the DB instance, but resources outside of the VPC can't access it.

After you complete the tutorial, there will be a public and private subnet in each Availability Zone in your VPC. In one Availability Zone, the EC2 instance will be in the public subnet, and the DB instance will be in the private subnet.

> **Note**
>
> There's no charge for creating an AWS account. However, by completing this tutorial, you might incur costs for the AWS resources you use. You can delete these resources after you complete the tutorial if they are no longer needed.

The following diagram shows the configuration when accessibility is public.



Warning

We don't recommend using 0.0.0.0/0 for HTTP access, since you would make it possible for all IP addresses to access your public InfluxDB instance via HTTP. This approach is not acceptable even for a short time in a test environment. Authorize only a specific IP address or range of addresses to access your InfluxDB instances using HTTP for web UI or API access.

This tutorial creates a DB instance running InfluxDB with the AWS Management Console. We will focus only on the DB instance size and DB instance identifier. We will use the default settings for the other configuration options. The DB instance created by this example will be private.

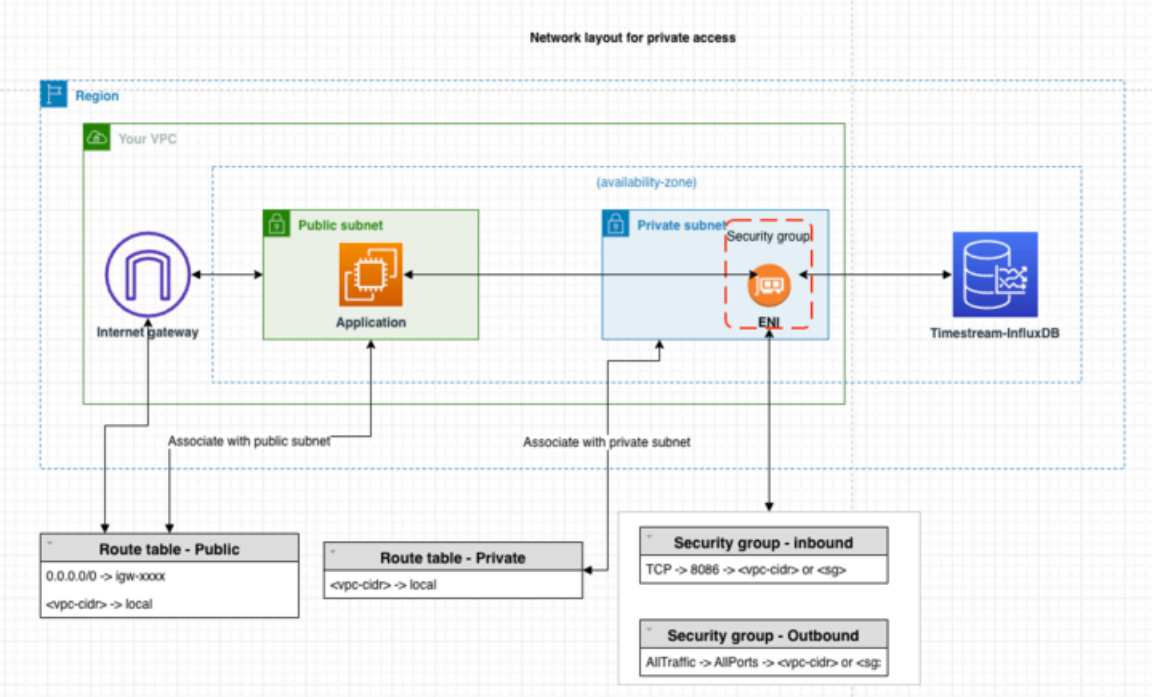
Other settings that you could configure include availability, security, and logging. To create a public DB instance, you must choose to make your instance **Publicly accessible** on the **Connectivity configuration** section. For information about creating DB instances, see Creating a DB instance.

If your instance is not publicly accessible, do the following:

- Create a host on the VPC of the instance through which you can tunnel traffic.
- Set up SSH tunneling to the instance. For more information, see Amazon EC2 instance port forwarding with AWS Systems Manager.
- In order for the certificate to work, add the following line to the `/etc/hosts` file of your client machine: `127.0.0.1`. This is the DNS address of your instance.
- Connect to your instance using the fully qualified domain name, for example, *https://<DNS>:8086*.

> **Note**
>
> Localhost is unable to validate the certificate because localhost is not part of the certificate SAN.

The following diagram shows the configuration when accessibility is private:



Prerequisites

Before you begin, complete the steps in the following sections:

- Sign up for an AWS account.
- Create an administrative user.

Step 1: Create an Amazon EC2 instance

Create an Amazon EC2 instance that you will use to connect to your database.

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you want to create the EC2 instance.
3. Choose **EC2 Dashboard**, and then choose **Launch instance**.
4. When the **Launch an instance** page opens, choose the following settings:
   a. Under **Name and tags**, enter `ec2-database-connect` for **Name**.
   b. Under **Application and OS Images (Amazon Machine Image)**, choose **Amazon Linux**, and then select **Amazon Linux 2023 AMI**. Keep the default selections for the other choices.
   c. Under **Instance type**, choose **t2.micro**.
   d. Under **Key pair (login)**, choose a **Key pair name** to use an existing key pair. To create a new key pair for the Amazon EC2 instance, choose **Create new key pair** and then use the **Create key pair** window to create it. For more information about creating a new key pair, see Create a key pair for your Amazon EC2 instance in the *Amazon Elastic Compute Cloud User Guide*.
   e. For **Allow SSH traffic from** in **Network settings**, choose the source of SSH connections to the EC2 instance. You can choose **My IP** if the displayed IP address is correct for SSH connections. Otherwise, you can determine the IP address to use to connect to EC2 instances in your VPC using Secure Shell (SSH). To determine your public IP address, in a different browser window or tab, you can use the service at checkip.amazonaws.com/. An example of an IP address is `192.0.2.1/32`. In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, make sure to determine the range of IP addresses used by client computers.

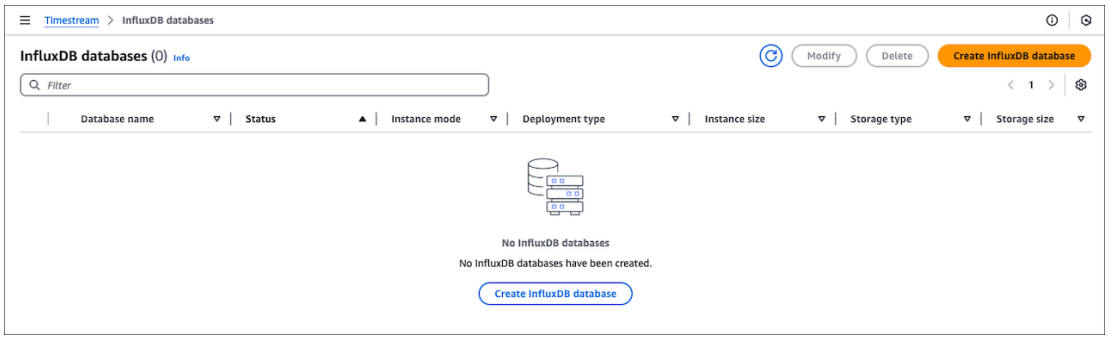
> **Warning**
>
> We do not recommend using 0.0.0.0/0 for SSH access, since you would make it possible for all IP addresses to access your public EC2 instances using SSH. This approach is not acceptable even for a short time in a test environment. Authorize only a specific IP address or range of addresses to access your EC2 instances using SSH.

Step 2: Create an InfluxDB DB instance

The basic building block of Amazon Timestream for InfluxDB is the DB instance. This environment is where you run your InfluxDB databases.

In this example, you will create a DB instance running the InfluxDB database engine with a db.influx.large DB instance class.

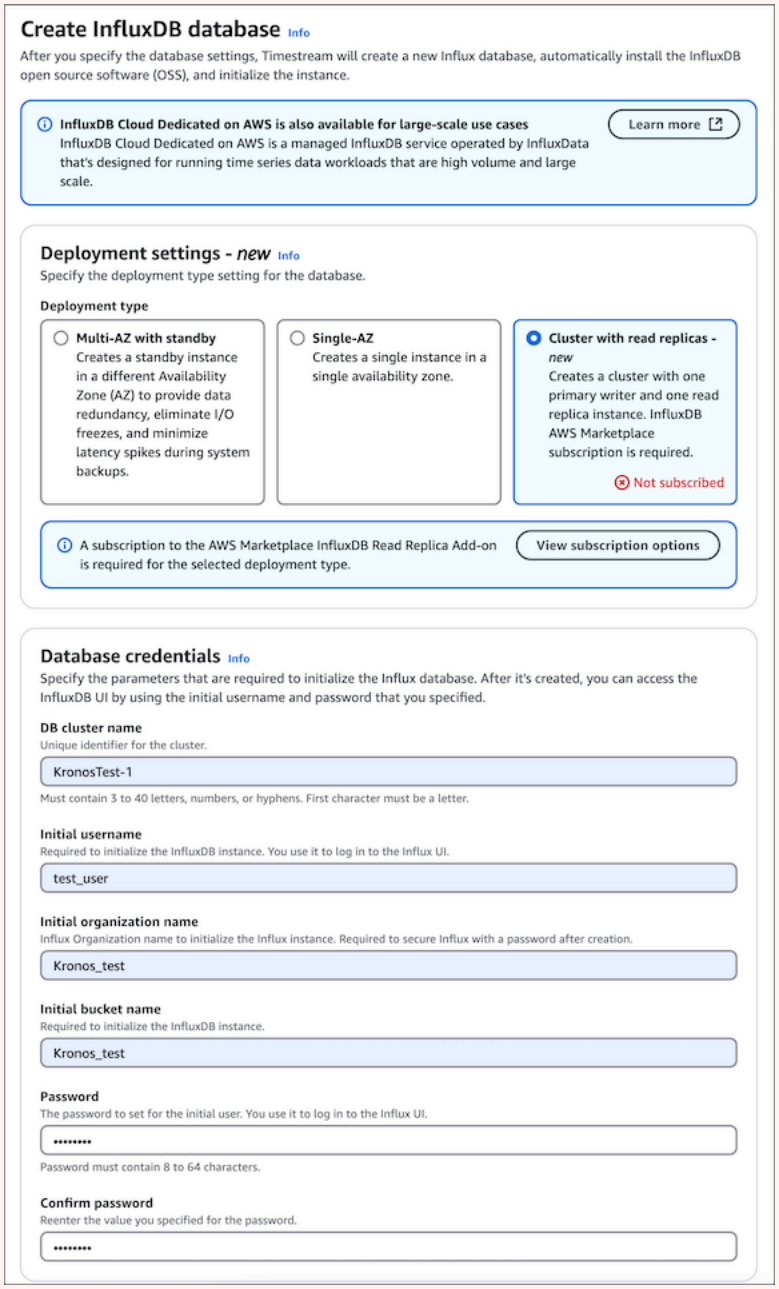
1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console at https://console.aws.amazon.com/timestream/.
2. In the upper-right corner of the Amazon Timestream for InfluxDB console, choose the AWS Region in which you want to create the DB instance.
3. In the navigation pane, choose **InfluxDB Databases**.
4. Choose **Create InfluxDB database**.



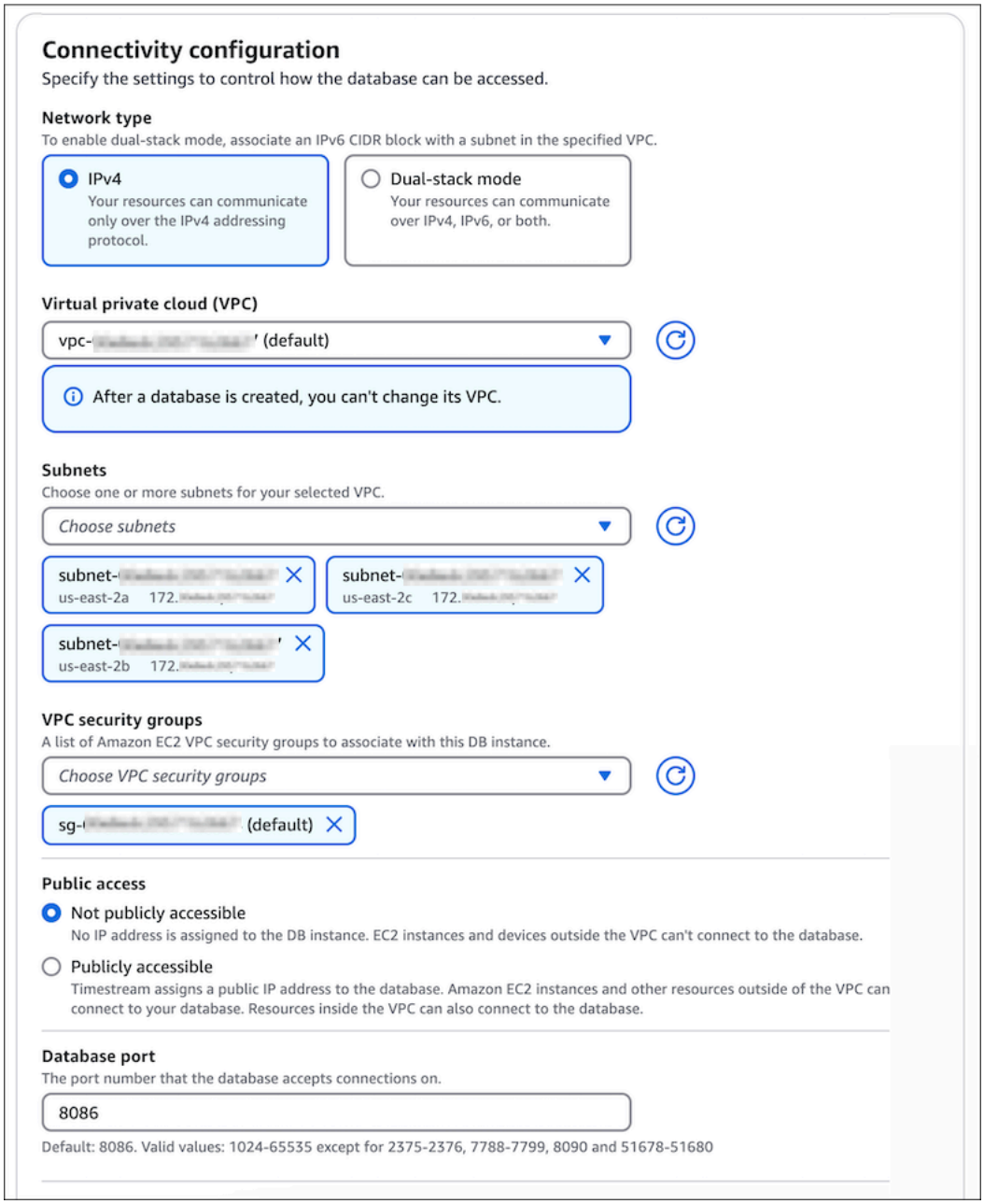
5. In the **Deployment settings** section, select **Cluster with read replicas**. Choose **View subscription options** to start a subscription for the read replica add-on. For more information, see Read replica licensing through AWS Marketplace.
6. In the **Database credentials** section, enter KronosTest-1 for **DB cluster name**.

7. Provide the InfluxDB basic configuration parameters: **Initial username**, **Initial organization name**, **Initial bucket name** and **Password**.

> **Important**
>
> You won't be able to view the user password again. You won't be able to access your instance and obtain an operator token without your password. If you don't record it, you might have to change it. See Creating a new operator token for your InfluxDB instance.
>
> If you need to change the user password after the DB instance is available, you can modify the DB instance to do so. For more information about modifying a DB instance, see Updating DB instances.



8. In the **Instance configuration** section, select the **db.influx.large** DB instance class.
9. In the **Storage configuration** section, select **Influx IO Included (3K)** for **Storage type**.
10. In the **Connectivity configuration** section, select **IPv4** for the **Network type**. Make sure your InfluxDB instance is in the same subnet as your newly created EC2 instance. Under **Public access**, select **Not publicly accessible** to make your DB instance private.



11. In the **Failover settings** and **Parameter group settings** sections, keep the default values.
12. Configure your logs in **Log delivery settings** and create tags (optional). For more information about logs, see Setup to view InfluxDB logs on Timestream Influxdb Instances. For more details about adding tags, see Adding tags and labels to resources.
13. Choose **Create InfluxDB database**.
14. In the **Databases** list, chose the name of your new InfluxDB instance to show its details. The DB instance has a status of **Creating** until it is ready to use.

You can connect to the DB instance when the status changes to **Available**. Depending on the DB instance class and the amount of storage, it can take up to 20 minutes before the new instance is available.

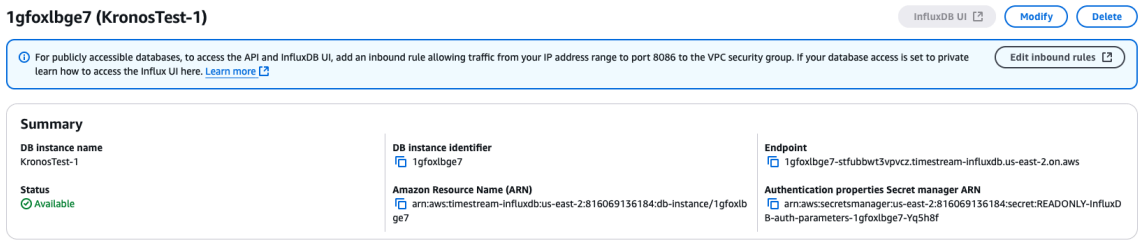
> **Important**
>
> At this time, you can't modify compute (instance types) and storage (storage types) configurations of existing instances.

Step 3: Access the InfluxDB UI

To access the InfluxDB UI from a private Timestream for InfluxDB DB instance, you must connect from within the same subnet and security group. One way to facilitate this connection is to create a bastion host within the private subnet.

A bastion host is a special-purpose server that acts as a secure entry point to critical systems, protecting your network from external access. It serves as a gateway between your secure internal network and the outside world.

> **Note**
>
> For publicly accessible Timestream for InfluxDB DB instances, you can access the InfluxDB UI via the **InfluxDB UI** button on the instance details page in the console. Note that this button will be disabled for instances that are not publicly accessible.
> If you have a public DB instance, connect to the InfluxDB UI via the console and proceed to Step 4: Send Telegraf data to your InfluxDB instance.



Follow these steps to create and configure your bastion host:

1. **Create a bastion host:** To create a bastion host, you can launch a new EC2 instance or use an existing one. Ensure that the instance has the necessary network setup to access the security group you used to create the private Timestream for InfluxDB instance you are trying to access.
2. **Connect to the InfluxDB UI:** Once you have created a bastion host, you can use the endpoint displayed in the console to connect to the InfluxDB UI. The endpoint will be in the format `<db-identifier>-<*>.timestream-influxdb.<region>.on.aws`. In China, it will be `<db-identifier>-<*>.timestream-influxdb.<region>.on.amazonwebservices.com.cn`.
3. **Configure your bastion host for local forwarding:** To set up local forwarding, use the AWS Systems Manager (SSM) session manager. Run the following command, replacing *`bastion-ec2-instance-id`* with the ID of your bastion host instance, *`endpoint`* with the endpoint displayed in the above console, and *`port-number`* with the port number you want to use:

   ```
   aws ssm start-session --target bastion-ec2-instance-id \
   --document-name AWS-StartPortForwardingSessionToRemoteHost \
   --parameters '{"host":["endpoint"], "portNumber":["port-number"],
    "localPortNumber":["port-number"]}'
   ```

   You may be prompted to install the SessionManagerPlugin. For more details, see [Install the Session Manager plugin for the AWS CLI](#).
4. **Access the InfluxDB UI:** After completing the above steps, you can access the InfluxDB UI at http://localhost:*`port-number`*. You will need to acknowledge the "not secure" message.
5. **Enable domain name validation:** To enable domain name validation, add the following line to your `/etc/hosts` file (Linux), `/private/etc/hosts` (Mac), or `C:\Windows\System32\drivers\etc` (Windows).

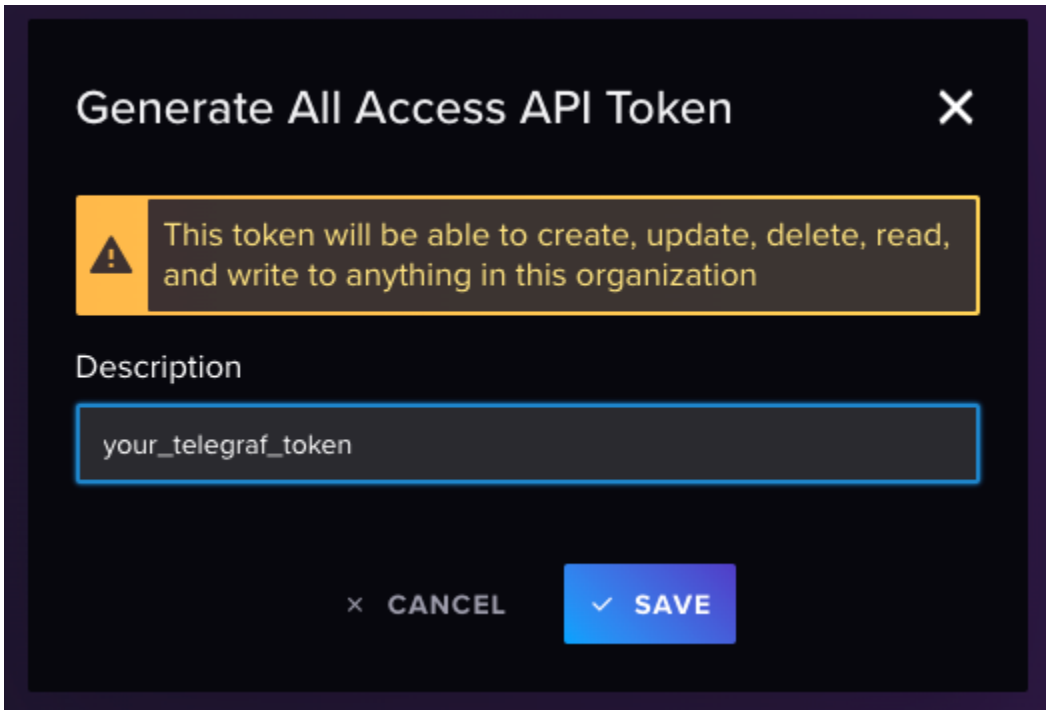
   ```
   127.0.0.1    endpoint
   ```
6. You can now access the InfluxDB UI using https://*`endpoint`*:*`port-number`*.

Step 4: Send Telegraf data to your InfluxDB instance

You can now start sending telemetry data to your InfluxDB DB instance using the Telegraf agent. In this example, you'll install and configure a Telegraf agent to send performance metrics to you InfluxDB DB instance.

1. After you connect to the InfluxDB UI, you should see a new browser window with a login prompt. Enter the credentials you used earlier to create your InfluxDB DB instance.
2. In the left navigation pane, click on the arrow icon and select **API Tokens**.
3. For this test, choose **Generate API Token**. Select **All Access API Token** from the dropdown list.

> **Note**
>
> For production scenarios, we recommend creating tokens with specific access to the required buckets that are built for specific Telegraf needs.



4. Your token will appear on the screen.

> **Important**
>
> Make sure to copy and save the token since it will not be displayed again.

5. Connect to the EC2 instance that you created earlier by following the steps in Connect to your Linux instance using SSH in the *Amazon Elastic Compute Cloud User Guide*.

We recommend that you connect to your EC2 instance using SSH. If the SSH client utility is installed on Windows, Linux, or Mac, you can connect to the instance using the following command format:

```
ssh -i location_of_pem_file ec2-user@ec2-instance-public-dns-name
```

For example, assume that `ec2-database-connect-key-pair.pem` is stored in `/dir1` on Linux, and the public IPv4 DNS for your EC2 instance is `ec2-12-345-678-90.compute-1.amazonaws.com`. Your SSH command would look as follows:

```
ssh -i /dir1/ec2-database-connect-key-pair.pem ec2-
user@ec2-12-345-678-90.compute-1.amazonaws.com
```

6. Get the latest version of Telegraf installed on your instance. To do this, use the following command:

```
cat <<EOF | sudo tee /etc/yum.repos.d/influxdata.repo
[influxdata]
name = InfluxData Repository - Stable
baseurl = https://repos.influxdata.com/stable/\$basearch/main
enabled = 1
gpgcheck = 1
gpgkey = https://repos.influxdata.com/influxdata-archive_compat.key
EOF

sudo yum install telegraf
```

7. Configure your Telegraf instance.

> **Note**
>
> If telegraf.conf does not exist or it does not contain a `timestream` section, you can generate one with:
>
> ```
> telegraf –section-filter agent:inputs:outputs –input-filter cpu:mem –output-
> filter timestream config > telegraf.conf
> ```

a. Edit the configuration file usually located at `/etc/telegraf`.

```
sudo nano /etc/telegraf/telegraf.conf
```

b. Configure the input plugins for CPUs, memory metrics, and disk usage.

```
[[inputs.cpu]]
  percpu = true
  totalcpu = true
  collect_cpu_time = false
  report_active = false

[[inputs.mem]]

[[inputs.disk]]
  ignore_fs = ["tmpfs", "devtmpfs", "devfs"]
```

c. Configure the output plugin to send data to your InfluxDB DB instance and save your changes.

```
[[outputs.influxdb_v2]]
   urls = ["https://us-west-2-1.aws.cloud2.influxdata.com"]
   token = "<your_telegraf_token"
   organization = "your_org"
   bucket = "your_bucket"
   timeout = "5s"
```

d. Configure the Timestream target.

```
# Configuration for sending metrics to Amazon Timestream.
[[outputs.timestream]]

  ## Amazon Region and credentials
  region = "us-east-1"
  access_key = "<AWS key here>"
  secret_key = "<AWS secret key here>"
  database_name = "<timestream database name>" # needs to exist

  ## Specifies if the plugin should describe on start.
  describe_database_on_start = false
  mapping_mode = "multi-table" # allows multiple tables for each input metrics
```

```
create_table_if_not_exists = true
create_table_magnetic_store_retention_period_in_days = 365
create_table_memory_store_retention_period_in_hours = 24

use_multi_measure_records = true # Important to use multi-measure records
measure_name_for_multi_measure_records = "telegraf_measure"
max_write_go_routines = 25
```

8. Enable and start the Telegraf service.

```
$ sudo systemctl enable telegraf
$ sudo systemctl start telegraf
```

Step 5: Delete the Amazon EC2 instance and the InfluxDB DB instance

After you explore the Telegraf-generated data using your your InfluxDB DB instance with the InfluxDB UI, delete both your EC2 and your InfluxDB DB instances so you are no longer charged for them.

To delete the EC2 instance:

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, choose **Instances**.
3. Select the checkbox next to the EC2 instance's name, and then select **Instance state**. Choose **Terminate (delete) instance**.
4. Choose **Terminate (delete)** when prompted for confirmation.

For more information about deleting an EC2 instance, see Terminate Amazon EC2 instances in the *Amazon Elastic Compute Cloud User Guide*.

To delete the DB instance with no final DB snapshot:

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console at https://console.aws.amazon.com/timestream/.
2. In the navigation pane, choose **InfluxDB databases**.
3. Select the DB instance you want to delete. Choose **Delete**

4. Confirm the deletion and choose **Delete**.

Creating a new operator token for your InfluxDB instance

If you need to get the Operator Token for your new InfluxDB instance, perform the following steps:

1. To change your operator token, we recommend using the Influx CLI. For instructions, please see: Install and use the Influx CLI.
2. Configure your CLI to use `--username-password` to be able to create the operator:

```
influx config create --config-name CONFIG_NAME1  --host-url "https://
yourinstanceid.eu-central-1.timestream-influxdb.amazonaws.com:8086" --org [YOURORG]
  --username-password [YOURUSERNAME] --active
```

3. Create your new operator token. You will be asked for your password to confirm this step.

```
influx auth create --org [YOURORG] --operator
```

> **Important**
>
> Once a new operator token has been created, you will need to update any client that is currently using the old one.

Migrating data from self-managed InfluxDB to Timestream for InfluxDB

The Influx migration script is a Python script that migrates data between InfluxDB OSS instances, whether those instances are managed by AWS or not.

InfluxDB is a time series database. InfluxDB contains *points*, which contain a number of key-value pairs and a timestamp. When points are grouped by key-value pairs, they form a series. A series is grouped by a string identifier called a *measurement*. InfluxDB is often used for operations monitoring, IOT data, and analytics. A *bucket* is a kind of container within InfluxDB to store data. AWS-managed InfluxDB is InfluxDB within the AWS ecosystem. InfluxDB provides the InfluxDB v2 API for accessing data and making changes to the database. The InfluxDB v2 API is what the Influx migration script uses to migrate data.

- The Influx migration script can migrate buckets and their metadata, migrate all buckets from all organizations, or do a full migration, which replaces all data on the destination instance.
- The script backups data from the source instance locally, on whatever system executes the script, then restores the data to the destination instance. The data is kept in code>influxdb-backup-<timestamp></timestamp> directories, one for each migration.
- The script provides a number of options and configurations including mounting S3 buckets to limit local storage usage during migration and choosing which organizations to use during migration.

Topics

- Preparation
- How to use scripts
- Migration Overview

Preparation

Data migration for InfluxDB is accomplished with a Python script that utilizes InfluxDB CLI features and the InfluxDB v2 API. Execution of the migration script requires the following environment configuration:

- **Supported Versions:** A minimum version of 2.3 of InfluxDB and Influx CLI is supported.
- **Token Environment Variables**
  - Create the environment variable `INFLUX_SRC_TOKEN` containing the token for your source InfluxDB instance.
  - Create the environment variable `INFLUX_DEST_TOKEN` containing the token for your destination InfluxDB instance.
- **Python 3**
  - Check installation: `python3 --version`.
  - If not installed, install from the Python website. Minimum version 3.7 required. On Windows the default Python 3 alias is simply python.
  - The Python module requests is required. Install it with: `shell python3 -m pip install requests`
  - TThe Python module influxdb_client is required. Install it with: `shell python3 -m pip install influxdb_client`

- **InfluxDB CLI**
  - Confirm installation: `influx version`.
  - If not installed, follow the installation guide in the InfluxDB documentation.

    Add influx to your $PATH.
- **S3 Mounting Tools (Optional)**

  When S3 mounting is used, all backup files are stored in a user-defined S3 bucket. S3 mounting can be useful to save space on the executing machine or when backup files need to be shared. If S3 mounting isn't used, by omitting the `--s3-bucket` option, then a local `influxdb-backup-<millisecond timestamp>` directory will be created to store backup files in the same directory that the script was run.

  For Linux: mountpoint-s3.

  For Windows: rclone (Prior rclone configuration is needed).
- **Disk Space**
  - The migration process automatically creates unique directories to store sets of backup files and retains these backup directories in either S3 or on the local filesystem, depending on the program arguments provided.
  - Ensure there is enough disk space for database backup, ideally double the size of the existing InfluxDB database if you choose to omit the `--s3-bucket` option and use local storage for backup and restoration.
  - Check space with `df -h (UNIX/Linux)` or by checking drive properties on Windows.
- **Direct Connection**

  Ensure a direct network connection exists between the system running the migration script and the source and destination systems. `influx ping --host <host>` is one way to verify a direct connection.

How to use scripts

A simple example of running the script is the command:

```
python3 influx_migration.py --src-host <source host> --src-bucket <source bucket> --
dest-host <destination host>
```

Which migrates a single bucket.

All options can be viewed by running:

```
python3 influx_migration.py -h
```

Usage

```
shell   influx_migration.py [-h] [--src-bucket SRC_BUCKET] [--dest-bucket DEST_BUCKET]
 [--src-host SRC_HOST] --dest-host DEST_HOST [--full] [--confirm-full] [--src-org
 SRC_ORG] [--dest-org DEST_ORG] [--csv] [--retry-restore-dir RETRY_RESTORE_DIR] [--dir-
name DIR_NAME] [--log-level LOG_LEVEL] [--skip-verify] [--s3-bucket S3_BUCKET]
```

Options

- **-confirm-full** (optional): Using `--full` without `--csv` will replace all tokens, users, buckets, dashboards, and any other key-value data in the destination database with the tokens, users, buckets, dashboards, and any other key-value data in the source database. `--full` with `--csv` only migrates all bucket and bucket metadata, including bucket organizations. This option (`--confirm-full`) will confirm a full migration and proceed without user input. If this option is not provided, and `--full` has been provided and `--csv` not provided, then the script will pause for execution and wait for user confirmation. This is a critical action, proceed with caution. Defaults to false.
- **-csv** (optional): Whether to use csv files for backing up and restoring. If `--full` is passed as well then all user-defined buckets in all organizations will be migrated, not system buckets, users, tokens, or dashboards. If a singular organization is desired for all buckets in the destination server instead of their already-existing source organizations, use `--dest-org`.
- **-dest-bucket DEST_BUCKET** (optional): The name of the InfluxDB bucket in the destination server, must not be an already existing bucket. Defaults to value of `--src-bucket` or None if `--src-bucket` not provided.
- **-dest-host DEST_HOST**: The host for the destination server. Example: http://localhost:8086.
- **-dest-org DEST_ORG** (optional): The name of the organization to restore buckets to in the destination server. If this is omitted, then all migrated buckets from the source server will retain their original organization and migrated buckets may not be visible in the destination server without creating and switching organizations. This value will be used in all forms of restoration whether a single bucket, a full migration, or any migration using csv files for backup and restoration.

- **-dir-name DIR_NAME** (optional): The name of the backup directory to create. Defaults to `influxdb-backup-<timestamp>`. Must not already exist.
- **-full** (optional): Whether to perform a full restore, replacing all data on destination server with all data from source server from all organizations, including all key-value data such as tokens, dashboards, users, etc. Overrides `--src-bucket` and `--dest-bucket`. If used with `--csv`, only migrates data and metadata of buckets. Defaults to false.
- **h, --help**: Shows help message and exits.
- **-log-level LOG_LEVEL**(optional): The log level to be used during execution. Options are debug, error, and info. Defaults to info.
- **-retry-restore-dir RETRY_RESTORE_DIR**(optional): Directory to use for restoration when a previous restore failed, will skip backup and directory creation, will fail if the directory doesn't exist, can be a directory within an S3 bucket. If a restoration fails, the backup directory path that can be used for restoration will be indicated relative to the current directory. S3 buckets will be in the form `influxdb-backups/<s3 bucket>/<backup directory>`. The default backup directory name is `influxdb-backup-<timestamp>`.
- **-s3-bucket S3_BUCKET**(optional): The name of the S3 bucket to use to store backup files. On Linux this is simply the name of the S3 bucket, such as `amzn-s3-demo-bucket1`, given `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables have been set or `${HOME}/.aws/credentials` exists. On Windows, this is the `rclone` configured remote and bucket name, such as `my-remote:amzn-s3-demo-bucket1`. All backup files will be left in the S3 bucket after migration in a created `influxdb-backups-<timestamp>` directory. A temporary mount directory named `influx-backups` will be created in the directory from where this script is ran. If not provided, then all backup files will be stored locally in a created `influxdb-backups-<timestamp>` directory from where this script is run.
- **-skip-verify**(optional): Skip TLS certificate verification.
- **-src-bucket SRC_BUCKET**(optional): The name of the InfluxDB bucket in the source server. If not provided, then `--full` must be provided.
- **-src-host SRC_HOST**(optional): The host for the source server. Defaults to http://localhost:8086.

As noted previously, `mountpoint-s3` and `rclone` are needed if `--s3-bucket` is to be used, but can be ignored if the user doesn't provide a value for `--s3-bucket`, in which case backup files will be stored in a unique directory locally.

Migration Overview

After meeting the prerequisites:

1. Run Migration Script: Using a terminal app of your choice, run the Python script to transfer data from the source InfluxDB instance to the destination InfluxDB instance.
2. Provide Credentials: Provide host addresses and ports as CLI options.
3. Verify Data: Ensure the data is correctly transferred by:
   a. Using the InfluxDB UI and inspecting buckets.
   b. Listing buckets with `influx bucket list -t <destination token> --host <destination host address> --skip-verify`.
   c. Using `influx v1 shell -t <destination token> --host <destination host address> --skip-verify` and running `SELECT * FROM <migrated bucket>.<retention period>.<measurement name> LIMIT 100` to view contents of a bucket or `SELECT COUNT(*) FROM <migrated bucket>.<retention period>.<measurment name>` to verify the correct number of records have been migrated.

Example run

1. Open a terminal app of your choice and make sure the required prerequisites are properly installed:

```
~ > python3 --version
Python 3.11.5
~ > influx version
Influx CLI 2.7.3 (git: 8b962c7e75) build_date: 2023-04-28T14:22:49Z
~ > s3fs --version
Amazon Simple Storage Service File System V1.92 (commit:unknown) with GnuTLS(gcrypt)
Copyright (C) 2010 Randy Rizun <rrizun@gmail.com>
License GPL2: GNU GPL version 2 <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
~ >
```

2. Navigate to the migration script:

```
~ > cd sample-code/influxdb-sample/migration/influxdb
~/sample-code/influxdb-sample/migration/influxdb > ls *.py
influx_migration.py
~/sample-code/influxdb-sample/migration/influxdb >
```

3. Prepare the following information:

   a. Name of the source bucket to be migrated.

   b. (Optional) Choose a new bucket name for the migrated bucket in the destination server.

   c. Root token for source and destination influx instances.

   d. Host address of source and destination influx instances.

   e. (Optional) S3 bucket name and credentials; AWS Command Line Interface credentials should be set in the OS environment variables.

```
# AWS credentials (for timestream testing)
    export AWS_ACCESS_KEY_ID="xxx"
    export AWS_SECRET_ACCESS_KEY="xxx"
```

   f. Construct the command as:

```
python3 influx_migration.py --src-bucket [amzn-s3-demo-source-bucket]  --dest-
bucket [amzn-s3-demo-destination-bucket] --src-host [source host] --dest-host
 [dest host] --s3-bucket [amzn-s3-demo-bucket2](optional) --log-level debug
```

   g. Execute the script:

```
~/sample-code/influxdb-sample/migration/influxdb > python3 influx_migration.py --src-bucket primary-bucket --src-host $INFLUXDB_1_HOST --dest-host $KRO
NOS_HOST --dest-bucket new-bucket-name
```

   h. Wait for the script to finish executing.

   i. Check the newly migrated bucket for data integrity, `performance.txt`. This file, located under the same directory where the script was run, contains some basic information on how long each step took.

Migration scenarios

Example 1: Simple Migration Using Local Storage

You want to migrate a single bucket, amzn-s3-demo-primary-bucket, from the source server (`http://localhost:8086`) to a destination server (`http://dest-server-address:8086`).

After ensuring you have TCP access (for HTTP access) to both machines hosting the InfluxDB instances on port 8086 and you have both source and destination tokens and have stored them as the environment variables `INFLUX_SRC_TOKEN` and `INFLUX_DEST_TOKEN`, respectively, for added security:

```
python3 influx_migration.py --src-bucket amzn-s3-demo-primary-bucket --src-host http://
localhost:8086 --dest-host http://dest-server-address:8086
```

The output should look similar to the following:

```
INFO: influx_migration.py: Backing up bucket data and metadata using the InfluxDB CLI
2023/10/26 10:47:15 INFO: Downloading metadata snapshot
2023/10/26 10:47:15 INFO: Backing up TSM for shard 1
2023/10/26 10:47:15 INFO: Backing up TSM for shard 8245
2023/10/26 10:47:15 INFO: Backing up TSM for shard 8263
[More shard backups . . .]
2023/10/26 10:47:20 INFO: Backing up TSM for shard 8240
2023/10/26 10:47:20 INFO: Backing up TSM for shard 8268
2023/10/26 10:47:20 INFO: Backing up TSM for shard 2
INFO: influx_migration.py: Restoring bucket data and metadata using the InfluxDB CLI
2023/10/26 10:47:20 INFO: Restoring bucket "96c11c8876b3c016" as "amzn-s3-demo-primary-
bucket"
2023/10/26 10:47:21 INFO: Restoring TSM snapshot for shard 12772
2023/10/26 10:47:22 INFO: Restoring TSM snapshot for shard 12773
[More shard restores . . .]
2023/10/26 10:47:28 INFO: Restoring TSM snapshot for shard 12825
2023/10/26 10:47:28 INFO: Restoring TSM snapshot for shard 12826
INFO: influx_migration.py: Migration complete
```

The directory `influxdb-backup-<timestamp>` will be created and stored in the directory from where the script was run, containing backup files.

Example 2: Full Migration Using Local Storage and Debug Logging

Same as above except you want to migrate all buckets, tokens, users, and dashboards, deleting the buckets in the destination server, and proceeding without user confirmation of a complete database migration by using the `--confirm-full` option. You also want to see what the performance measurements are so you enable debug logging.

```
python3 influx_migration.py --full --confirm-full --src-host http://localhost:8086 --
dest-host http://dest-server-address:8086 --log-level debug
```

The output should look similar to the following:

```
INFO: influx_migration.py: Backing up bucket data and metadata using the InfluxDB CLI
2023/10/26 10:55:27 INFO: Downloading metadata snapshot
2023/10/26 10:55:27 INFO: Backing up TSM for shard 6952
2023/10/26 10:55:27 INFO: Backing up TSM for shard 6953
[More shard backups . . .]
2023/10/26 10:55:36 INFO: Backing up TSM for shard 8268
2023/10/26 10:55:36 INFO: Backing up TSM for shard 2
DEBUG: influx_migration.py: backup started at 2023-10-26 10:55:27 and took 9.41 seconds
 to run.
INFO: influx_migration.py: Restoring bucket data and metadata using the InfluxDB CLI
2023/10/26 10:55:36 INFO: Restoring KV snapshot
2023/10/26 10:55:38 WARN: Restoring KV snapshot overwrote the operator token, ensure
 following commands use the correct token
2023/10/26 10:55:38 INFO: Restoring SQL snapshot
2023/10/26 10:55:39 INFO: Restoring TSM snapshot for shard 6952
2023/10/26 10:55:39 INFO: Restoring TSM snapshot for shard 6953
[More shard restores . . .]
2023/10/26 10:55:49 INFO: Restoring TSM snapshot for shard 8268
2023/10/26 10:55:49 INFO: Restoring TSM snapshot for shard 2
DEBUG: influx_migration.py: restore started at 2023-10-26 10:55:36 and took 13.51
 seconds to run.
INFO: influx_migration.py: Migration complete
```

Example 3: Full Migration Using CSV, Destination Organization, and S3 Bucket

Same as the previous example but using Linux or Mac and storing the files in the S3 bucket, `amzn-s3-demo-bucket`. This avoids backup files overloading the local storage capacity.

```
python3 influx_migration.py --full --src-host http://localhost:8086 --dest-host http://
dest-server-address:8086 --csv --dest-org MyOrg --s3-bucket amzn-s3-demo-bucket
```

The output should look similar to the following:

```
INFO: influx_migration.py: Creating directory influxdb-backups
INFO: influx_migration.py: Mounting amzn-s3-demo-influxdb-migration-bucket
```

```
INFO: influx_migration.py: Creating directory influxdb-backups/amzn-s3-demo-bucket/
influxdb-backup-1698352128323
INFO: influx_migration.py: Backing up bucket data and metadata using the InfluxDB v2
 API
INFO: influx_migration.py: Restoring bucket data and metadata from csv
INFO: influx_migration.py: Restoring bucket amzn-s3-demo-some-bucket
INFO: influx_migration.py: Restoring bucket amzn-s3-demo-another-bucket
INFO: influx_migration.py: Restoring bucket amzn-s3-demo-primary-bucket
INFO: influx_migration.py: Migration complete
INFO: influx_migration.py: Unmounting influxdb-backups
INFO: influx_migration.py: Removing temporary mount directory
```

Configuring a DB instance

This section shows how to set up your Amazon Timestream for InfluxDB DB instance. Before creating a DB instance, decide on the DB instance class that will run the DB instance. Also, decide where the DB instance will run by choosing an AWS Region. Next, create the DB instance.

You can configure a DB instance with a DB parameter group.A DB parameter group acts as a container for engine configuration values that are applied to one or more DB instances.

The parameters that are available depend on the DB engine and DB engine version. You can specify a DB parameter group when you create a DB instance. You can also modify a DB instance to specify them.

> **Important**
>
> At this time, you can't modify compute (Instance types) and Storage (Storage Types) configuration of existing instances.

Creating a DB instance

Using the console

1. Sign in to the AWS Management Console and open Amazon Timestream for InfluxDB.
2. In the upper-right corner of the Amazon Timestream for InfluxDB console, choose the AWS Region in which you want to create the DB instance.
3. In the navigation pane, choose **InfluxDB Databases**.

4. Choose **Create Influx database**.
5. For **DB Instance Identifier**. enter a name that will identify your instance.
6. Provide the InfluxDB basic configuration parameters **User Name, Organization, Bucket Name and Password**.

   > **Important**
   >
   > Your user name, organization, bucket name and password will be stored as a secret in AWS Secrets Manager that will be created for your account.

   If you need to change the user password after the DB instance is available, you can modify using the Influx CLI.
7.
8. For **DB Instance Class**, select an instance size that better fit your workload needs.
9. For **DB Storage Class**, select a storage class that fits your need. In all cases, you will only need to configure the allocated storage.
10. In the **Connectivity configuration** section, make sure your InfluxDB instance is in the same subnet as your new the clients that require connectivity to your Timestream for InfluxDB DB instance. You could also chose to make your DB instance publicly available.
11. Choose **Create Influx database**.
12. In the **Databases** list, choose the name of your new InfluxDB instance to show its details. The DB Instance has a status of **Creating** until is ready to use.
13. When the status changes to **Available**, you can connect to the DB instance. Depending on the DB instance class and the amount of storage, it can take up to 20 minutes before the new instance is available.

Using the CLI

To create a DB instance by using the AWS Command Line Interface, call the `create-db-instance` command with the following parameters:

```
--name
--vpc-subnet-ids
--vpc-security-group-ids
```

```
--db-instance-type
--db-storage-type
--username
--organization
--password
--allocated-storage
```

For information about each setting, see Settings for DB instances.

Example: Using default engine configs

For Linux, macOS, or Unix:

```
aws timestream-influxdb create-db-instance \
    --name myinfluxDbinstance \
    --allocated-storage 400 \
    --db-instance-type db.influx.4xlarge \
    --vpc-subnet-ids subnetid1 subnetid2
    --vpc-security-group-ids mysecuritygroup \
    --username masterawsuser \
    --password \
    --db-storage-type InfluxIOIncludedT2
```

For Windows:

```
aws timestream-influxdb create-db-instance \
    --name myinfluxDbinstance \
    --allocated-storage 400 \
    --db-instance-type db.influx.4xlarge \
    --vpc-subnet-ids subnetid1 subnetid2
    --vpc-security-group-ids mysecuritygroup \
    --username masterawsuser \
    --password \
    --db-storage-type InfluxIOIncludedT2
```

Using the API

To create a DB instance by using the AWS Command Line Interface, call the `CreateDBInstance` command with the following parameters:

For information about each setting, see Settings for DB instances.

> **Important**
>
> Part of the DBInstance response object you receive an influxAuthParametersSecretArn. This will hold an ARN to a SecretsManager secret in your account. It will only be populated after your InfluxDB DB instances is available. The secret contains influx authentication parameters provided during the `CreateDbInstance` process. This is a READONLY copy as any updates/modifications/deletions to this secret doesn't impact the created DB instance. If you delete this secret, our API response will still refer to the deleted secret ARN.

Once you have finished creating your Timestream for InfluxDB DB instance, we recommend you download, install and configure the Influx CLI.

The influx CLI provides a simple way to interact with InfluxDB from a command line. For detailed installation and setup instructions, see Use the Influx CLI.

Settings for DB instances

You can create a DB instance using the console, the `create-db-instance` CLI command, or the `CreateDBInstance` Timestream for InfluxDB API operation.

The following table provides details about settings that you choose when you create a DB instance.

Console Setting	Description	CLI option and Timestream API parameter
Allocated storage	The amount of storage to allocate for your DB instance (in gibibytes). In some cases, allocating a higher amount of storage for your DB instance than the size of your database can improve I/O performan ce. For more information, see InfluxDB instance storage.	CLI: `allocated -storage` API: `allocated storage`
Bucket Name	A name for the bucket to initialize the InfluxDb instance	CLI: `bucket` API: `bucket`

Console Setting	Description	CLI option and Timestream API parameter
DB instance type	The configuration for your DB instance. For example, a db.influx.large DB instance class has 16 GiB memory, 2 vCPUs, memory optimized. If possible, choose a DB instance type large enough that a typical query working set can be held in memory. When working sets are held in memory, the system can avoid writing to disk, which improves performance. For more information, see DB instance class types.	CLI: `db-instance-type` API: `Dbinstancetype`
DB instance identifier	The name for your DB instance. Name your DB instances in the same way that you name your on-premises servers. Your DB instance identifier can contain up to 63 alphanumeric characters, and must be unique for your account in the AWS Region you chose.	CLI: `db-instance-identifier` API: `Dbinstanceidentifier`
DB parameter group	A parameter group for your DB instance. You can choose the default parameter group, or you can create a custom parameter group. For more information, see Working with DB parameter groups..	CLI: `db-parameter-group-name` API: `DBParameterGroupName`
Log Delivery Setting	The name of the S3 bucket were the InfluxDB logs will be stored.	CLI: `LogDeliveryConfiguration` API: `log-delivery-configuration`

Console Setting	Description	CLI option and Timestream API parameter
Multi-AZ deployment	Create a standby instance to create a passive secondary replica of your DB instance in another Availability Zone for failover support. We recommend Multi-AZ for production workloads to maintain high availability. For development and testing, you can choose Do not create a standby instance. For more information, see Configuring and managing a multi-AZ deployment.	CLI: `MultiAz` API: `multi-az`
Network Type	The IP addressing protocols supported by the DB instance. IPv4 (the default) to specify that resources can communicate with the DB instance only over the Internet Protocol version 4 (IPv4) addressing protocol. Dual-stack mode to specify that resources can communicate with the DB instance over IPv4, Internet Protocol version 6 (IPv6), or both. Use dual-stack mode if you have any resources that must communicate with your DB instance over the IPv6 addressing protocol. Also, make sure that you associate an IPv6 CIDR block with all subnets in the DB subnet group that you specify. While IPv6 is public by default, we do support private IPv6 endpoints, keep in mind that this is a one way door since we do not support changing the *Publicly Accessible* flag after instance creation.	CLI: `network-type` API: `NetworkType`

Console Setting	Description	CLI option and Timestream API parameter
Password	This will be your master use password use to Initialize your InfluxDB Db instance. You will use this password to log in into the InfluxUI to obtain your operator token.	CLI: `password` API: `password`
Public Access	Yes to give the DB instance a public IP address, meaning that it's accessible outside the VPC. To be publicly accessible, the DB instance also has to be in a public subnet in the VPC. No to make the DB instance accessible only from inside the VPC. To connect to a DB instance from outside of its VPC, the DB instance must be publicly accessible. Also, access must be granted using the inbound rules of the DB instance's security group. In addition, other requirements must be met.	CLI: `publicly-accessible` API: `PubliclyAccessible`
Storage Type	The storage type for your DB instance You can choose between 3 different types Provisioned influx IOPS Included storage according to your workloads requirements: * Influx IOPS Included 3000 IOPS * Influx IOPS Included 12000 IOPS * INflux IOPS Included 16000 IOPS For more information, see [InfluxDB instance storage].	CLI: `db-storage-type` API: `DbStorageType`

Console Setting	Description	CLI option and Timestream API parameter
Initial username	This will be the master user to initialize your InfluxDB DB instance with. You will use this username to log in into the InfluxUI to obtain your operator token.	CLI: `username` API: `Username`
Subnets	A vpc subnet to associate with this DB instance.	CLI: `vpc-subnet-ids` API: `VPCSubnetIds`
VPC Security Group (firewall)	The security group to associate with the DB instance.	CLI: `vpc-security-group-ids` API: `VPCSecurityGroupIds`

Connecting to an Amazon Timestream for InfluxDB DB instance

Before you can connect to a DB instance, you must create the DB instance. For information, see Creating a DB instance. After Amazon Timestream provisions your DB instance, use the InfluxDB API, influx CLI, or any compatible client or utility for InfluxDB to connect to the DB instance.

Topics

- Finding the connection information for an Amazon Timestream for InfluxDB DB instance
- Database authentication options
- Working with parameter groups

Finding the connection information for an Amazon Timestream for InfluxDB DB instance

The connection information for a DB instance includes its endpoint, port, username, password, and a valid access token, such as the operator or all-access token. For example, for a Timestream for InfluxDB DB instance, suppose that the endpoint value is `c5vasdqn0b-3ksj4dla5nfjhi.timestream-influxdb.us-east-1.on.aws`. In this case, the port value is 8086, and the database user is *admin*. Given this information, to access the instance you will use:

- The endpoint of your instance, `c5vasdqn0b-3ksj4dla5nfjhi.timestream-influxdb.us-east-1.on.aws:8086`.
- Either the username and password supplied when creating the instance or valid access token.

Instances created before December 9, 2024 will have an endpoint that contains the instance name instead of the instance ID. For example: `influxdb1-123456789.us-east-1.timestream-influxdb.amazonaws.com`.

> **Important**
>
> As part of the DB instance response object, you will receive a `influxAuthParametersSecretArn`. This will hold an ARN to a Secrets Manager secret in your account. t will only be populated after your InfluxDB DB instances are available. The secret contains Influx authentication parameters provided during the `CreateDbInstance` process. This is a **read-only** copy as any updates/modifications/deletions to this secret doesn't impact the created DB instance. If you delete this secret, our API response will still refer to the deleted secret ARN.

The endpoint is unique for each DB instance, and the values of the port and user can vary. To connect to a DB instance, you can use the influx CLI, InfluxDB API, or any client compatible with InfluxDB.

To find the connection information for a DB instance, use the AWS Management Console. You can also use the AWS Command Line Interface (AWS CLI) `describe-db-instances` command or the Timestream for InfluxDB API `GetDBInstance` operation.

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream console.
2. In the navigation pane, choose **InfluxDB Databases** to display a list of your DB instances.
3. Choose the name of the DB instance to display its details.
4. In the **Summary** section, copy the endpoint. Also, note the port number. You will need both the endpoint and the port number to connect to the DB instance.

If you need to find the username and password information, choose the **Configuration Details** tab and choose the `influxAuthParametersSecretArn` to access your Secrets Manager.

Using the CLI

- To find the connection information for a InfluxDB DB instance by using the AWS CLI, call the `get-db-instance` command. In the call, query for the DB instance ID, endpoint, port, and influxAuthParametersSecretArn.

  For Linux, macOS, or Unix:

  ```
  aws timestream-influxdb get-db-instance --identifier id \
   --query "[name,endpoint,influxAuthParametersSecretArn]"
  ```

  For Windows:

  ```
  aws timestream-influxdb get-db-instance --identifier id ^
   --query "[name,endpoint,influxAuthParametersSecretArn]"
  ```

  Your output should be similar to the following. To access the username information, you will need to check the `InfluxAuthParameterSecret`.

  ```
  [
      [
          "mydb",
          "mydbid-123456789012.timestream-influxdb.us-east-1.on.aws",
          8086,
      ]
  ]
  ```

Creating access tokens

With this information, you are going to be able to connect to your instance to retrieve or create your access tokens. There are several ways to achieve this:

Using the CLI

1. If you haven't already, download, install, and configure the influx CLI.
2. When configuring your influx CLI config, use `--username-password` to authenticate.

```
influx config create --config-name YOUR_CONFIG_NAME --host-url "https://
yourinstanceid-accountidentifier.timestream-influxdb.us-east-1.on.aws:8086" --org
 yourorg --username-password admin --active
```

3. Use the influx auth create command to re-create your operator token. Take into account that this process will invalidate the old operator token.

```
influx auth create --org kronos --operator
```

4. Once you have the operator token, you can use the influx auth list command to view all your tokens. You can use the influx auth create command to create an all-access token.

> **Important**
>
> You will need to perform this step to obtain your operator token first. Then you will be able to create new tokens using the InfluxDB API or CLI.

Using the InfluxDB UI

1. Browse to your Timestream for InfluxDB instance using the created endpoint to log in and access the InfluxDB UI. You will need to use the username and password used to create your InfluxDB DB instance. You can retrieve this information from the `influxAuthParametersSecretArn` that was specified in the response object of the `CreateDbInstance`.

   Alternatively you can open the InfluxDB UI from the Amazon Timestream for InfluxDB console:

   a. Sign in to the AWS Management Console and open the Timestream for InfluxDB console at https://console.aws.amazon.com/timestream/.
   b. In the upper-right corner of the Amazon Timestream for InfluxDB console, choose the AWS Region in which you created the DB instance.
   c. In the **Databases** list, choose the name of your InfluxDB instance to show its details. In the upper right corner, choose **InfluxDB UI**.
2. Once logged in to your InfluxDB UI, navigate to **Load Data** and then **API Tokens** using the left navigation bar.
3. Choose **Generate API Token** and select **All Access API Token**.
4. Enter a description for the API token and choose **SAVE**.
5. Copy the generated token and store it for safe keeping.

> **Important**
>
> When creating tokens from the InfluxDB UI, the newly created tokens are only going to be shown once. Make sure you copy these. Otherwise, you will need to re-create them.

Using the InfluxDB API

- Send a request to the InfluxDB API `/api/v2/authorizations` endpoint using the POST request method.

  Include the following with your request:

  a. Headers:

     i. Authorization: Token <INFLUX_OPERATOR_TOKEN>
     ii. Content-Type: application/json

  b. Request body: JSON body with the following properties:

     i. status: "active"
     ii. description: API token description
     iii. orgID: InfluxDB organization ID

iv. permissions: Array of objects where each object represents permissions for an InfluxDB resource type or a specific resource. Each permission contains the following properties:

A. action: “read” or “write”

B. resource: JSON object that represents the InfluxDB resource to grant permission to. Each resource contains at least the following property: orgID: InfluxDB organization ID

C. type: Resource type. For information about what InfluxDB resource types exist, use the /api/v2/resources endpoint.

The following example uses `curl` and the InfluxDB API to generate an all-access token:

```
export INFLUX_HOST=https://instanceid-123456789.timestream-influxdb.us-east-1.on.aws
export INFLUX_ORG_ID=<YOUR_INFLUXDB_ORG_ID>
export INFLUX_TOKEN=<YOUR_INFLUXDB_OPERATOR_TOKEN>

curl --request POST \
"$INFLUX_HOST/api/v2/authorizations" \
  --header "Authorization: Token $INFLUX_TOKEN" \
  --header "Content-Type: text/plain; charset=utf-8" \
  --data '{
    "status": "active",
    "description": "All access token for get started tutorial",
    "orgID": "'"$INFLUX_ORG_ID"'",
    "permissions": [
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "authorizations"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "authorizations"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "buckets"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "buckets"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "dashboards"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "dashboards"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type": "orgs"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type": "orgs"}},
```

```
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"sources"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"sources"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type": "tasks"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"tasks"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"telegrafs"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"telegrafs"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type": "users"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"users"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"variables"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"variables"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"scrapers"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"scrapers"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"secrets"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"secrets"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"labels"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"labels"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type": "views"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"views"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"documents"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"documents"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"notificationRules"}},
    {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"notificationRules"}},
    {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
"notificationEndpoints"}},
```

```
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "notificationEndpoints"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "checks"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "checks"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type": "dbrp"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type": "dbrp"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "notebooks"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "notebooks"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "annotations"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "annotations"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "remotes"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "remotes"}},
      {"action": "read", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "replications"}},
      {"action": "write", "resource": {"orgID": "'"$INFLUX_ORG_ID"'", "type":
 "replications"}}
    ]
  }
'
```

Database authentication options

Amazon Timestream for InfluxDB supports the following ways to authenticate database users:

- **Password authentication** – Your DB instance performs all administration of user accounts. You create users, specify passwords, and administer tokens using the InfluxDB UI, influx CLI, or InfluxDB API.
- **Token authentication** – Your DB instance performs all administration of user accounts. You can create users, specify passwords, and administer tokens using your operator token via the influx CLI and InfluxDB API.

Encrypted connections

You can use Secure Socket Layer (SSL) or Transport Layer Security (TLS) from your application to encrypt a connection to a DB instance. The certificates needed for the TLS handshake between InfluxDB and the applications created and managed by the Kronos service. When the certificate is renewed, the instance is automatically updated with the latest version without requiring any user intervention.

Working with parameter groups

Database parameters specify how the database is configured. For example, database parameters can specify the amount of resources, such as memory, to allocate to a database.

You manage your database configuration by associating your DB instances with parameter groups. Amazon Timestream for InfluxDB defines parameter groups with default settings. You can also define your own parameter groups with customized settings.

Overview of parameter groups

A DB parameter group acts as a container for engine configuration values that are applied to one or more DB instances.

Topics

- Default and custom parameter groups
- Creating a DB parameter group
- Static and dynamic DB instance parameters
- Supported parameters and parameter values

Default and custom parameter groups

DB instances use DB parameter groups. The following sections describe configuring and managing DB instance parameter groups.

Creating a DB parameter group

You can create a new DB parameter group using the AWS Management Console, the AWS Command Line Interface, or the Timestream API.

The following limitations apply to the DB parameter group name:

- The name must be 1 to 255 letters, numbers, or hyphens.
- Default parameter group names can include a period, such as `default.InfluxDB.2.7`. However, custom parameter group names can't include a period.
- The first character must be a letter.
- The name cannot start with “dbpg-”
- The name can't end with a hyphen or contain two consecutive hyphens.
- If you create a DB instance without specifying a DB parameter group, the DB instance uses the InfluxDB engine defaults.

You can't modify the parameter settings of a default parameter group. Instead, you can do the following:

1. Create a new parameter group.
2. Change the settings of your desired parameters. Not all DB engine parameters in a parameter group are eligible to be modified.
3. Update your DB instance to use the custom parameter group. For information about updating a DB instance, see Updating DB instances.

> **Note**
>
> If you have modified your DB instance to use a custom parameter group, and you start the DB instance, Amazon Timestream for InfluxDB automatically reboots the DB instance as part of the startup process.
> Currently, you won’t be able to modify custom parameter groups once they have been created. If you need to change a parameter, it is required that you create a new custom parameter group and assign it to the instances that require this configuration change.
> If you update an existing DB instance to assign a new parameter group, it will always be applied immediately and reboot your instance.

Static and dynamic DB instance parameters

InfluxDB DB instance parameters are always static. They behave as follows:

When you change a static parameter, save the DB parameter group, and assign it to an instance, the parameter change takes effect automatically after the instance is rebooted.

When you associate a new DB parameter group with a DB instance, Timestream applies the modified static parameters only after the DB instance is rebooted. Currently the only option is apply immediately.

For more information about changing the DB parameter group, see Updating DB instances.

Supported parameters and parameter values

To determine the supported parameters for your DB instance, view the parameters in the DB parameter group used by the DB instance. For more information, see Viewing parameter values for a DB parameter group.

For more information about all parameters supported by the open-source version of InfluxDB, see InfluxDB configuration options. Currently you will only be able to modify the following InfluxDB parameters:

Parameter	Description	Default value	Value	Valid range	Note
flux-log-enabled	Include option to show detailed logs for Flux queries	FALSE	Boolean	N/A	
log-level	Log output level. InfluxDB outputs log entries with severity levels greater than or equal to the level specified.	info	debug, info, error	N/A	
no-tasks	Disable the task	FALSE	Boolean	N/A	

Parameter	Description	Default value	Value	Valid range	Note
	scheduler. If problematic tasks prevent InfluxDB from starting, use this option to start InfluxDB without scheduling or executing tasks.				
query-concurrency	Number of queries allowed to execute concurrently. Setting to 0 allows an unlimited number of concurrent queries.	0		0 to 256	

Parameter	Description	Default value	Value	Valid range	Note
query-queue-size	Maximum number of queries allowed in execution queue. When queue limit is reached, new queries are rejected. Setting to 0 allows an unlimited number of queries in the queue.	1,024		N/A	
tracing-type	Enable tracing in InfluxDB and specifies the tracing type. Tracing is disabled by default.	""	log, jaeger	N/A	

Parameter	Description	Default value	Value	Valid range	Note
metrics-disabled	Disable the HTTP / metrics endpoint which exposes internal InfluxDB metrics.	FALSE		N/A	

Parameter	Description	Default value	Value	Valid range	Note
http-idle-timeout	Maximum duration the server should keep established connections alive while waiting for new requests. Set to `0` for no timeout.	3m0s	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	Hours: -Minimum: 0 -Maximum: 256,205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685	

Parameter	Description	Default value	Value	Valid range	Note
http-read-header-timeout	Maximum duration the server should try to read HTTP headers for new requests. Set to `0` for no timeout.	10s	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	Hours: -Minimum: 0 -Maximum: 256,205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685	

Parameter	Description	Default value	Value	Valid range	Note
http-read-timeout	Maximum duration the server should try to read the entirety of new requests. Set to `0` for no timeout.	0	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	Hours: -Minimum: 0 -Maximum: 256,205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685	

Parameter	Description	Default value	Value	Valid range	Note
http-write-timeout	Maximum duration the server should spend processing and responding to write requests. Set to `0` for no timeout.	0	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	Hours: -Minimum: 0 -Maximum: 256,205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685	

Parameter	Description	Default value	Value	Valid range	Note
influxql-max-select-buckets	Maximum number of group by time buckets a SELECT statement can create. 0 allows an unlimited number of buckets.	0	Long	Minimum: 0 Maximum: 9,223,372,036,854,775,807	
influxql-max-select-point	Maximum number of points a SELECT statement can process. 0 allows an unlimited number of points. InfluxDB checks the point count every second (so queries exceeding the maximum aren't immediately aborted).	0	Long	Minimum: 0 Maximum: 9,223,372,036,854,775,807	

Parameter	Description	Default value	Value	Valid range	Note
influxql-max-select-series	Maximum number of series a `SELECT` statement can return. `0` allows an unlimited number of series.	0	Long	Minimum: 0 Maximum: 9,223,372,036,854,775,807	
pprof-disabled	Disable the `/debug/pprof` HTTP endpoint. This endpoint provides runtime profiling data and can be helpful when debugging.	TRUE	Boolean	N/A	While InfluxDB sets pprof-disabled as `false` by default, AWS sets it as `true` by default.
query-initial-memory-bytes	Initial bytes of memory allocated for a query.	0	Long	Minimum: 0 Maximum: query-memory-bytes	

Parameter	Description	Default value	Value	Valid range	Note
query-max-memory-bytes	Maximum total bytes of memory allowed for queries.	0	Long	Minimum: 0 Maximum: 9,223,372,036,854,775,807	
query-memory-bytes	Specifies the Time to Live (TTL) in minutes for newly created user sessions.	0	Long	Minimum: 0 Maximum: 2,147,483,647	Must be greater than or equal to query-initial-memory-bytes.
session-length	Specifies the Time to Live (TTL) in minutes for newly created user sessions.	60	Integer	Minimum: 0 Maximum: 2,880	

Parameter	Description	Default value	Value	Valid range	Note
session-renew-disabled	Disables automatically extending a user's session TTL on each request. By default, every request sets the session's expiration time to 5 minutes from now. When disabled, sessions expire after the specified session length and the user is redirected to the login page, even if recently active.	FALSE	Boolean	N/A	

Parameter	Description	Default value	Value	Valid range	Note
storage-cache-max-memory-size	Maximum size (in bytes) a shard's cache can reach before it starts rejecting writes.	1,073,741,824	Long	Minimum: 0 Maximum: 549,755,813,888	Must be lower than instance's total memory capacity. We recommend setting it to below 15 percent of the total memory capacity.
storage-cache-snapshot-memory-size	Size (in bytes) at which the storage engine will snapshot the cache and write it to a TSM file to make more memory available.	26,214,400	Long	Minimum: 0 Maximum: 549,755,813,888	Must be lower than storage-cache-max-memory-size.

Parameter	Description	Default value	Value	Valid range	Note
storage-cache-snapshot-write-cold-duration	Duration at which the storage engine will snapshot the cache and write it to a new TSM file if the shard hasn't received writes or deletes.	10m0s	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	Hours: -Minimum: 0 -Maximum: 256,205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685	

Parameter	Description	Default value	Value	Valid range	Note
storage-compact-full-write-cold-duration	Duration at which the storage engine will compact all TSM files in a shard if it hasn't received writes or deletes.	4h0m0s	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	Hours: -Minimum: 0 -Maximum: 256,205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685	
storage-compact-throughput-burst	Rate limit (in bytes per second) that TSM compactions can write to disk.	50,331,648	Long	Minimum: 0 Maximum: 9,223,372,036,854,775,807	

Parameter	Description	Default value	Value	Valid range	Note
storage-max-concurrent-compactions	Maximum number of full and level compactions that can run concurrently. A value of `0` results in 50 percent of `runtime.GOMAXPROCS(0)` used at runtime. Any number greater than zero limits compactions to that value. This setting does not apply to cache snapshotting.	0	Integer	Minimum: 0 Maximum: 64	

Parameter	Description	Default value	Value	Valid range	Note
storage-max-index-log-file-size	Size (in bytes) at which an index write-ahead log (WAL) file will compact into an index file. Lower sizes will cause log files to be compacted more quickly and result in lower heap usage at the expense of write throughput.	1,048,576	Long	Minimum: 0 Maximum: 9,223,372,036,854,775,807	
storage-no-validate-field-size	Skip field size validation on incoming write requests.	FALSE	Boolean	N/A	

Parameter	Description	Default value	Value	Valid range	Note
storage-retention-check-interval	Interval of retention policy enforcement checks.	30m0s	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	N/A	Hours: -Minimum: 0 -Maximum: 256,205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685

Parameter	Description	Default value	Value	Valid range	Note
storage-series-file-max-concurrent-snapshot-compactions	Maximum number of snapshot compactions that can run concurrently across all series partitions in a database.	0	Integer	Minimum: 0 Maximum: 64	

Parameter	Description	Default value	Value	Valid range	Note
storage-series-id-set-cache-size	Size of the internal cache used in the TSI index to store previously calculated series results. Cached results are returned quickly rather than needing to be recalculated when a subsequent query with the same tag key/value predicate is executed. Setting this value to `0` will disable the cache and may decrease query performance.	100	Long	Minimum: 0 Maximum: 9,223,372,036,854,775,807	

Parameter	Description	Default value	Value	Valid range	Note
storage-wal-max-concurrent-writes	Maximum number writes to the WAL directory to attempt at the same time.	0	Integer	Minimum: 0 Maximum: 256	
storage-wal-max-write-delay	Maximum amount of time a write request to the WAL directory will wait when the the maximum number of concurrent active writes to the WAL directory has been met. Set to `0` to disable the timeout.	10m	Duration with unit `hours`, `minutes`, `seconds`, `milliseconds` . Example: `durationType=minutes,value=10`	Hours: -Minimum: 0 -Maximum: 256205 Minutes: -Minimum: 0 -Maximum: 15,372,286 Seconds: -Minimum: 0 -Maximum: 922,337,203 Milliseconds: -Minimum: 0 -Maximum: 922,337,203,685	

Parameter	Description	Default value	Value	Valid range	Note
ui-disabled	Disable the InfluxDB user interface (UI). The UI is enabled by default.	FALSE	Boolean	N/A	

Improperly setting parameters in a parameter group can have unintended adverse effects, including degraded performance and system instability. Always be cautious when modifying database parameters. Test parameter group setting changes on a test DB instance before applying those parameter group changes to a production DB instance.

Working with DB parameter groups

DB instances use DB parameter groups. The following sections describe configuring and managing DB instance parameter groups.

Topics

- Creating a DB parameter group
- Associating a DB parameter group with a DB instance
- Listing DB parameter groups
- Viewing parameter values for a DB parameter group

Creating a DB parameter group

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **Parameter groups**.
3. Choose **Create parameter group**.
4. In the **Parameter group name** box, enter the name of the new DB parameter group.
5. In the **Description** box, enter a description for the new DB parameter group.

6. Choose the parameters to modify and apply the desired values. For more information on supported parameters, see Supported parameters and parameter values.
7. Choose **Create parameter group**.

Using the AWS Command Line Interface

- To create a DB parameter group by using the AWS CLI, call the `create-db-parameter-group` command with the following parameters:

```
--db-parameter-group-name <value>
--description <value>
--endpoint_url <value>
--region <value>
--parameters (list) (string)
```

Example

For information about each setting, see Settings for DB instances. This example uses default engine configs.

```
aws timestream-influxdb create-db-parameter-group
    --db-parameter-group-name YOUR_PARAM_GROUP_NAME \
    --endpoint-url YOUR_ENDPOINT \
    --region YOUR_REGION \
    --parameters
 "InfluxDBv2={logLevel=debug,queryConcurrency=10,metricsDisabled=true}" \" \
    --debug
```

Associating a DB parameter group with a DB instance

You can create your own DB parameter groups with customized settings. You can associate a DB parameter group with a DB instance using the AWS Management Console, the AWS Command Line Interface, or the Timestream for InfluxDB API. You can do so when you create or modify a DB instance.

For information about creating a DB parameter group, see Creating a DB parameter group. For information about creating a DB instance, see Creating a DB instance. For information about modifying a DB instance, see Updating DB instances.

> **Note**
>
> When you associate a new DB parameter group with a DB instance, the modified static parameters are applied only after the DB instance is rebooted. Currently, only apply immediately is supported. Timestream for InfluxDB only support static parameters.

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**, and then choose the DB instance that you want to modify.
3. Choose **Update**. The **Update DB instance** page appears.
4. Change the **DB parameter group** setting.
5. Choose **Continue** and check the summary of modifications.
6. Currently only **Apply immediately** is supported. This option can cause an outage in some cases since it will reboot your DB instance.
7. On the confirmation page, review your changes. If they are correct, choose **Update DB instance** to save your changes and apply them. Or choose **Back** to edit your changes or **Cancel** to cancel your changes.

Using the AWS Command Line Interface

For Linux, macOS, or Unix:

```
aws timestream-influxdb update-db-instance
--identifier YOUR_DB_INSTANCE_ID \
--region YOUR_REGION \
--db-parameter-group-identifier YOUR_PARAM_GROUP_ID \
--log-delivery-configuration "{\"s3Configuration\": {\"bucketName\":
 \"${LOGGING_BUCKET}\", \"enabled\": false }}"
```

For Windows:

```
aws timestream-influxdb update-db-instance
```

```
--identifier YOUR_DB_INSTANCE_ID ^
--region YOUR_REGION ^
--db-parameter-group-identifier YOUR_PARAM_GROUP_ID ^
--log-delivery-configuration "{\"s3Configuration\": {\"bucketName\":
 \"${LOGGING_BUCKET}\", \"enabled\": false }}"
```

Listing DB parameter groups

You can list the DB parameter groups you've created for your AWS account.

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **Parameter groups**.
3. The DB parameter groups appear in a list.

Using the AWS Command Line Interface

To list all DB parameter groups for an AWS account, use the AWS Command Line Interface `list-db-parameter-groups` command.

```
aws timestream-influxdb list-db-parameter-groups --region region
```

To return a specific DB parameter groups for an AWS account, use the AWS Command Line Interface `get-db-parameter-group` command.

```
aws timestream-influxdb get-db-parameter-group --region region --identifier identifier
```

Viewing parameter values for a DB parameter group

You can get a list of all parameters in a DB parameter group and their values.

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.

2. In the navigation pane, choose **Parameter groups**.
3. The DB parameter groups appear in a list.
4. Choose the name of the parameter group to see its list of parameters.

Using the AWS Command Line Interface

To view the parameter values for a DB parameter group, use the AWS Command Line Interface `get-db-parameter-group` command. Replace *`parameter-group-identifier`* with your own information.

```
get-db-parameter-group --identifier parameter-group-identifier
```

Using the API

To view the parameter values for a DB parameter group, use the Timestream API `GetDbParameterGroup` command. Replace *`parameter-group-identifier`* with your own information.

```
GetDbParameterGroup parameter-group-identifier
```

Working with Multi-AZ read replica clusters for Amazon Timestream for InfluxDB

A read replica cluster deployment is an asynchronous deployment mode of Amazon Timestream for InfluxDB that allows you to configure read replicas attached to a primary DB instance. A read replica cluster has a writer DB instance and a reader DB instance in separate Availability Zones within the same AWS Region. Read replica clusters provide high availability and increased capacity for read workloads when compared to Multi-AZ DB instance deployments.

Instance class availability for read replica clusters

Read replica cluster deployments are supported for the same instance types as regular Timestream for InfluxDB instances.

Instance class	vCPU	Memory (GiB)	Storage type	Network bandwidth (Gbps)
db.influx .medium	1	8	Influx IOPS Included	10
db.influx.large	2	16	Influx IOPS Included	10
db.influx.xlarge	4	32	Influx IOPS Included	10
db.influx.2xlarge	8	64	Influx IOPS Included	10
db.influx.4xlarge	16	128	Influx IOPS Included	10
db.influx.8xlarge	32	256	Influx IOPS Included	12
db.influx .12xlarge	48	384	Influx IOPS Included	20
db.influx .16xlarge	64	512	Influx IOPS Included	25
db.influx .24xlarge	96	768	Influx IOPS Included	40

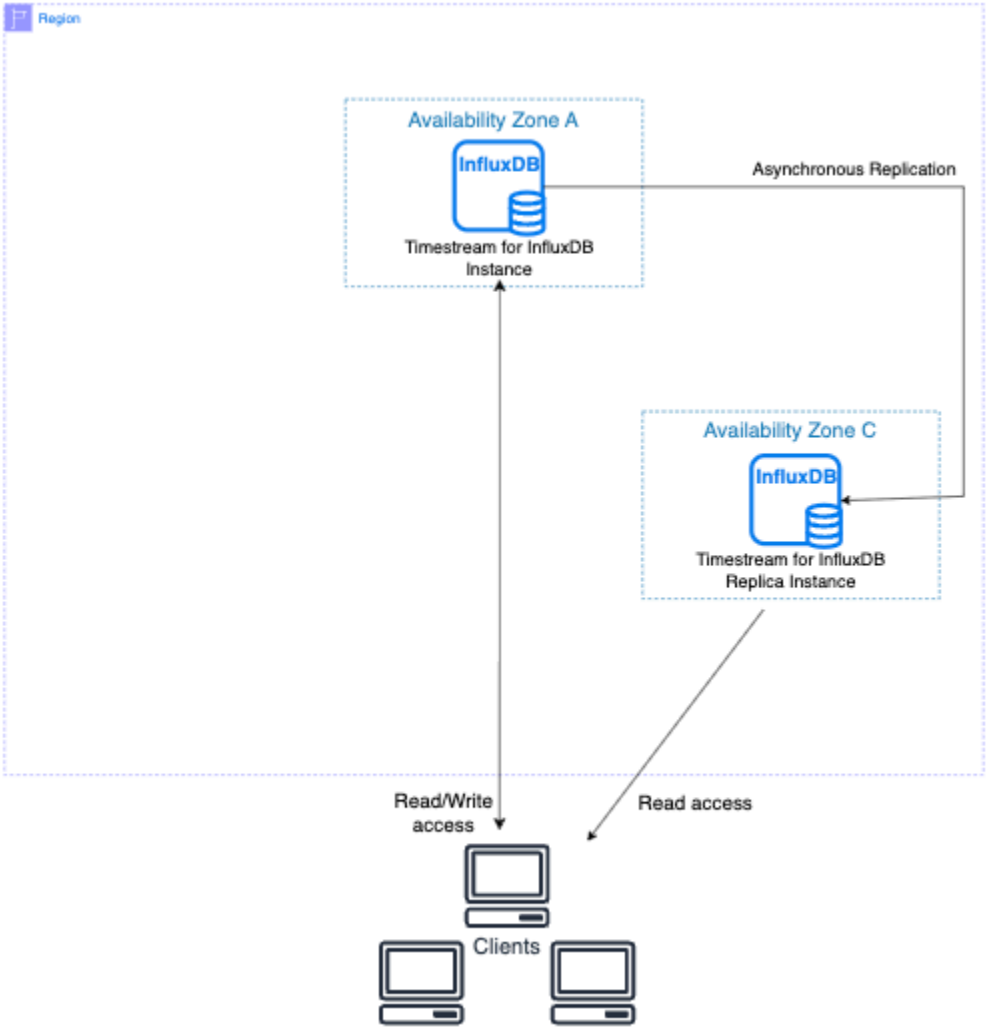
Read replica cluster architecture

With a read replica cluster, Amazon Timestream for InfluxDB automatically replicates all writes made to the writer DB instance to all reader DB instances using InfluxData's licensed read replica add-on. This replication is asynchronous and all writes are acknowledged as soon as they are committed by the writer node. Writes do not require acknowledgement from all reader nodes to be considered as a successful write. Once data is committed by the writer DB instance, it is replicated

to the read replica instance almost instantaneously. In case of unrecoverable writer failure, any data that has not been replicated over to at least one of the readers will be lost.

A read replica instance is a read-only copy of a writer DB instance. You can reduce the load on your writer DB instance by routing some or all of the queries from your applications to the read replica. In this way, you can elastically scale out beyond the capacity constraints of a single DB instance for read-heavy database workloads.

The following diagram shows a primary DB instance replicating to a read replica in a different Availability Zone. Clients have read/write access to the primary DB instance and read-only access to the replica.



Parameter groups for read replica clusters

In a read replica cluster, a *DB parameter group* acts as a container for engine configuration values that are applied to every DB instance in the read replica cluster. A default DB parameter group is

set based on the DB engine and DB engine version. The settings in the DB parameter group are used for all of the DB instances in the cluster.

When passing a specific DB parameter group using CreateDbCluster or UpdateDbCluster for Multi-AZ DB read replica, ensure the `storage-wal-max-write-delay` is set to a duration of 1 hour minimum. If no DB parameter group is specified, `storage-wal-max-write-delay` will default to 1 hour.

Replica lag in read replica clusters

Although Timestream for InfluxDB read replica clusters allow for high write performance, replica lag can still occur due to the nature of engine-based asynchronous replication. This lag can lead to potential data loss in the event of a failover, making it essential to monitor.

You can track the replica lag from CloudWatch by selecting **All metrics** in the AWS Management Console navigation pane. Choose **Timestream/InfluxDB**, then **By DbCluster**. Select your **DbClusterName** and then your **DbReaderInstanceName**. Here, besides the normal set of metrics tracked for all Timestream for InfluxDB instances (see below list), you will also see ReplicaLag, expressed in milliseconds.

- CPUUtilization
- MemoryUtilization
- DiskUtilization
- ReplicaLag (only for replica instance mode DB instances)

Common causes of replica lag

In general, replica lag occurs when the write and read workloads are too high for the reader DB instances to apply the transactions efficiently. Various workloads can incur temporary or continuous replica lag. Some examples of common causes are the following:

- High write concurrency or heavy batch updating on the writer DB instance, causing the apply process on the reader DB instances to fall behind.
- Heavy read workload that is using resources on one or more reader DB instances. Running slow or large queries can affect the apply process and can cause replica lag.
- Transactions that modify large amounts of data or DDL statements can sometimes cause a temporary increase in replica lag because the database must preserve commit order.

For a tutorial that shows you how to create a CloudWatch alarm when replica lag exceeds a set amount of time, see Tutorial: Create an Amazon CloudWatch alarm for Multi-AZ cluster replica lag for Amazon Timestream for InfluxDB.

Mitigating replica lag

For Timestream for InfluxDB read replica clusters, you can mitigate replica lag by reducing the load on your writer DB instance.

Availability and durability

Read replica clusters can be configured to either automatically fail over to one of the reader instances in case of writer failure to prioritize write availability or to avoid failing over to minimize tip data loss. Tip data refers to the replication gap of data not yet replicated to at least one of the reader nodes (see Replica lag in read replica clusters). The default and recommended behavior for read replica clusters is to automatically fail over in case of writer failures. However, if tip data loss is more important than write availability for your use cases, you can override the default by updating the cluster.

Read replica clusters ensure that all DB instances of the cluster are distributed across at least two Availability Zones to ensure increased write availability and data durability in case of an Availability Zone outage.

Topics

- Overview of Amazon Timestream for InfluxDB read replica clusters
- Creating a Timestream for InfluxDB read replica cluster
- Connecting to a Timestream for InfluxDB read replica DB cluster
- Modifying a read replica cluster for Amazon Timestream for InfluxDB
- Rebooting a read replica cluster in Amazon Timestream for InfluxDB
- Creating CloudWatch alarms to monitor Amazon Timestream for InfluxDB
- Read replica licensing through AWS Marketplace

Overview of Amazon Timestream for InfluxDB read replica clusters

The following sections discuss Timestream for InfluxDB read replica clusters:

Topics

- Use cases for read replicas
- How read replicas work
- Characteristics of Timestream for InfluxDB read replicas
- Read replica instance and storage types
- Considerations when deleting replicas

Use cases for read replicas

Using a read replica cluster might make sense in a variety of scenarios, including the following:

- Scaling beyond the compute or I/O capacity of a single DB instance for read-heavy database workloads. You can direct this excess read traffic to one or more read replicas.
- Serving read traffic while the primary writer instance is unavailable. In some cases, your primary DB instance might not be able to take I/O requests, for example, due to I/O suspension for backups or scheduled maintenance. In these cases, you can direct read traffic to your read replica. For this use case, keep in mind that the data on the read replica might be "stale" because the primary DB instance is unavailable. Also, keep in mind that you will need to turn off automatic failover for these scenarios to work.
- Business reporting or data warehousing scenarios where you might want business reporting queries to run against a read replica, rather than your production DB instance.
- Implementing disaster recovery. You can promote a read replica to primary as a disaster recovery solution if the primary DB instance fails.
- Faster failover for scenarios where availability is more important than durability. Since read replicas use asynchronous replication, there is a chance that some data that was committed by the primary writer instance was not replicated before a failover. However, for applications where uptime is paramount, this trade-off is acceptable. Depending on your workload characteristics, a failover to a read replica could be significantly faster than a failover to a standby DB instance that uses synchronous replication, as the replica instance is already running and does not need to start the engine. This can be particularly beneficial in use cases where every minute counts.

How read replicas work

To create a read replica cluster, Amazon Timestream for InfluxDB uses InfluxData’s licensed read replica add-ons. The add-on subscription is activated via the AWS Marketplace, directly from the

Amazon Timestream management console. For more details, see Read replica licensing through AWS Marketplace.

Read replicas are billed as standard DB instances at the same rates as the DB instance type used for each node in your cluster, plus the cost of InfluxData's licensed add-on. The cost of the add-on is billed in instance-hours via the AWS Marketplace. You aren't charged for the data transfer incurred in replicating data between the source DB instance and a read replica within the same AWS Region.

Once you have created and configured your read replica cluster and start accepting writes, Amazon Timestream for InfluxDB uses the asynchronous replication method to update the read replica whenever there is a change to the primary DB instance.

The read replica functions as a dedicated DB instance, exclusively accepting read-only connections. Applications can connect to a read replica in the same manner as they would to any other DB instance, providing a seamless and familiar experience. Amazon Timestream for InfluxDB automatically replicates all data from the primary DB instance to the read replica, ensuring data consistency and accuracy. Note that updates are done at the cluster level and applied at the same time to both the primary and replica.

Characteristics of Timestream for InfluxDB read replicas

Feature or behavior	Timestream for InfluxDB
What is the replicati on method?	Logical replication.
Can a replica be made writable?	No, Timestream for InfluxDB read replicas are designed to be read-only and cannot be made writable. While a read replica can be promoted to primary in the event of a failover, thereby accepting writes, at any given time, there can only be one writer DB instance in a Timestrea m for InfluxDB read replica cluster. This ensures data consistency and prevents conflicts that could arise from multiple writable instances. The read replica's role is to provide a redundant, read-only copy of the data, and it will automatically reject write requests to maintain data integrity .
Can backups be performed on the replica?	Yes, you can use the built-in engine capabilities to create backups using the Influx CLI.

Feature or behavior	Timestream for InfluxDB
Can you use parallel replication?	No, Timestream for InfluxDB has a single process handling replication.

Read replica instance and storage types

A read replica is created with the same instance and storage type as the primary DB instance. Any changes to the configuration must be made at the cluster level and will apply to all instances within the cluster. All instance and storage configurations available for Timestream for InfluxDB DB instances are available for Timestream for InfluxDB read replica clusters.

Instance types

Instance class	vCPU	Memory (GiB)	Storage type	Network bandwidth (Gbps)
db.influx .medium	1	8	Influx IOPS Included	10
db.influx.large	2	16	Influx IOPS Included	10
db.influx.xlarge	4	32	Influx IOPS Included	10
db.influx.2xlarge	8	64	Influx IOPS Included	10
db.influx.4xlarge	16	128	Influx IOPS Included	10
db.influx.8xlarge	32	256	Influx IOPS Included	12
db.influx .12xlarge	48	384	Influx IOPS Included	20

Instance class	vCPU	Memory (GiB)	Storage type	Network bandwidth (Gbps)
db.influx .16xlarge	64	512	Influx IOPS Included	25
db.influx .24xlarge	96	768	Influx IOPS Included	40

Storage options

Timestream for InfluxDB DB cluster storage	Source DB instance storage allocation	Included IOPS
Influx IO Included (3K)	20 GiB to 16 TiB	3,000 IOPS
Influx IO Included (12K)	400 GiB to 16 TiB	12,000 IOPS
Influx IO Included (16K)	400 GiB to 16 TiB	16,000 IOPS

Considerations when deleting replicas

If you no longer require read replicas, you can explicitly delete the cluster by calling the `delete-db-cluster` API. In the following example, replace each *`user input placeholder`* with your own information. Keep in mind that you cannot remove a single node from your cluster at this time.

```
aws timestream-influxdb delete-db-cluster \
            --region region \
            --endpoint endpoint \
            --db-cluster-id cluster-id
```

Creating a Timestream for InfluxDB read replica cluster

A Timestream for InfluxDB read replica cluster has a writer DB instance and a reader DB instance in separate Availability Zones. Timestream for InfluxDB read replica clusters provide high availability, increased capacity for read workloads, and faster failover when failover to replica is configured.

DB cluster prerequisites

> **Important**
>
> The following are prerequisites to complete before creating a read replica cluster.

Topics

- Configure the network for the DB cluster
- Additional prerequisites

Configure the network for the DB cluster

You can only create a Timestream for InfluxDB read replica DB cluster in a virtual private cloud (VPC) based on the Amazon VPC service. It must be in an AWS Region that has at least three Availability Zones. The DB subnet group that you choose for the DB cluster must cover at least three Availability Zones. This configuration ensures that each DB instance in the DB cluster is in a different Availability Zone.

To connect to your DB cluster from resources other than EC2 instances in the same VPC, configure the network connections manually.

Additional prerequisites

Before you create your read replica cluster, consider the following additional prerequisites:

To tailor the configuration parameters for your DB cluster, specify a DB cluster parameter group with the required parameter settings. For information about creating or modifying a DB cluster parameter group, see Parameter groups for read replica clusters.

Determine the TCP/IP port number to specify for your DB cluster. The firewalls at some companies block connections to the default ports. If your company firewall blocks the default port, choose another port for your DB cluster. All DB instances in a DB cluster use the same port.

Create a DB cluster

You can create a Timestream for InfluxDB read replica DB cluster using the AWS Management Console, the AWS CLI, or the Amazon Timestream for InfluxDB API.

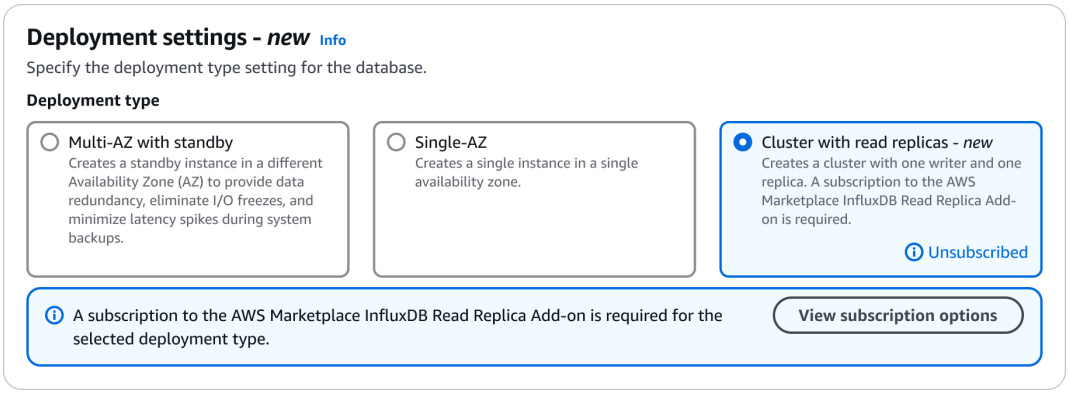
Using the AWS Management Console

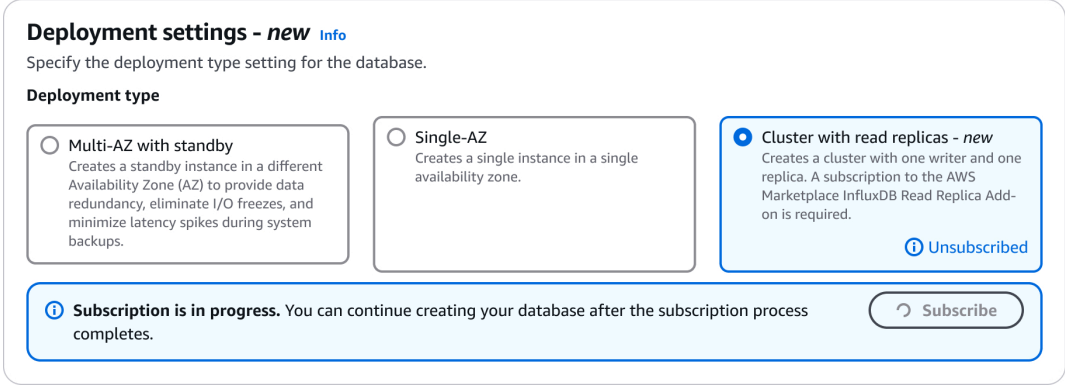
You can create a Timestream for InfluxDB read replica DB cluster by choosing **Cluster with read replicas** in the **Deployment settings** section.

To create a read replica DB cluster using the console:

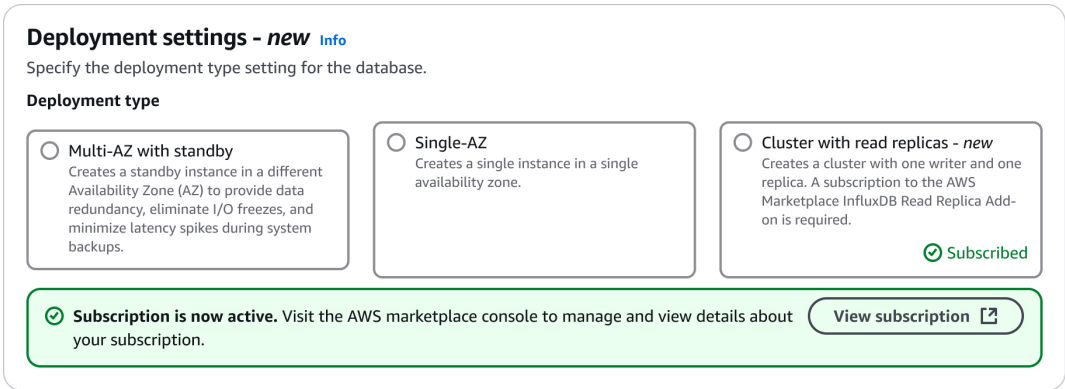
1. Sign in to the AWS Management Console and open the Amazon Timestream console.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you want to create the read replica DB cluster.
3. In the navigation pane, choose **InfluxDB databases**.
4. Choose **Create InfluxDB database**.
5. In **Deployment settings**, choose **Cluster with read replicas**.

   Once you select that option, a message will appear indicating you need to activate your subscription via the AWS Marketplace widget. Click on **View subscription options**. Note that it can take 1–2 minutes for the subscription to become active.

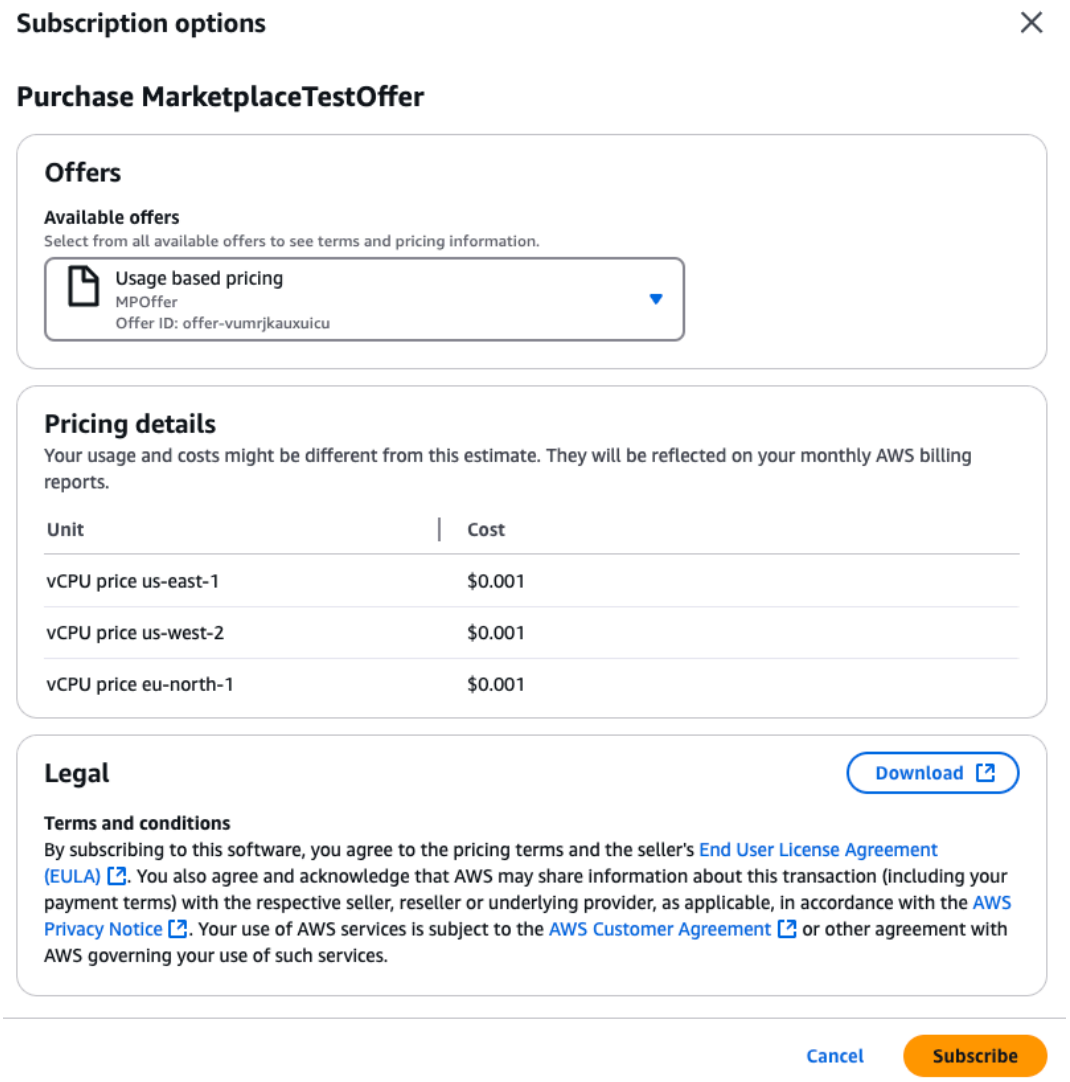




6. Once the subscription is active, click **View subscription**.



7. A window will appear presenting information on the cost per vCPU per instance hour for each Region. This follows the same compute pricing model where you are charged for the number of hours your instance is active based on the instance type you have selected. You will only need to subscribe to the add-on once, and that will allow you to create instances in all Regions where Timestream for InfluxDB is available.

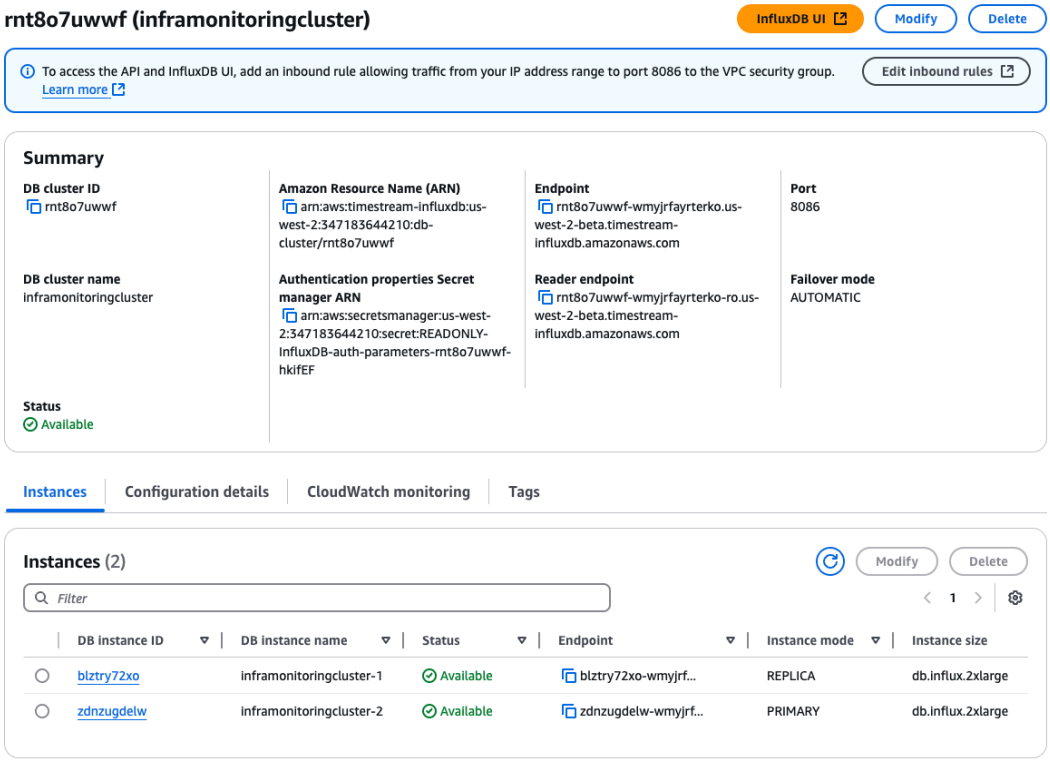


> **Important**
>
> To subscribe to the offer, you will need to have either AWSMarketplaceManageSubscriptions or AWSMarketplaceFullAccess permissions. For more information about these permissions, check Controlling access to AWS Marketplace subscriptions.

8. Once you confirm your subscription, the service will automatically select the Region based on the Region of your instance.
9. In **Database credentials**, complete the following fields:
   a. For **DB cluster name**, enter the identifier for your DB cluster.

b. Provide the InfluxDB basic initial configuration parameters: **username**, **organization name**, **bucket name**, and **password**.

10. In **Instance configuration**, specify the **DB instance class**. Select an instance size that best fits your workload needs. Keep in mind that this instance type will be used for all instances in your read replica DB cluster.
11. In **Storage configuration**, select a **Storage type** that fits your needs. In all cases, you will only need to configure the allocated storage. Keep in mind that this storage type will be used for all instances in your read replica DB cluster.
12. In the **Connectivity configuration** section, make sure your InfluxDB cluster is in the same subnet as the clients that require connectivity to your Timestream for InfluxDB DB instance. You could also choose to make your DB instance publicly available in the **Public access** subsection.
13. Choose **Create InfluxDB database**.
14. In the **InfluxDB databases** list, choose the name of your new InfluxDB cluster to show its details. The DB cluster will have a status of **Creating** until it is ready to use.
15. When the status changes to **Available**, you can connect to the DB cluster. Depending on the DB instance class and the amount of storage, it can take up to 20 minutes before the new instance is available.



16. Once created, you can click on your DB cluster identifier to retrieve information about your newly created cluster. The endpoint showing an instance mode of **PRIMARY** is the one you will need to use for writes and engine administration.

Using the AWS CLI

To create a DB instance using the AWS Command Line Interface, call the `create-db-cluster` command with the following parameters. Replace each *`user input placeholder`* with your own information.

```
aws timestream-influxdb create-db-cluster \
      --region region \
      --vpc-subnet-ids subnet-ids \
      --vpc-security-group-ids security-group-ids \
      --db-instance-type db.influx.large \
      --db-storage-type InfluxIOIncludedT2 \
      --allocated-storage 400 \
      --password password \
      --name cluster-name \
      --deployment-type MULTI_NODE_READ_REPLICAS \
      --publicly-accessible
      //--failover-mode is optional and defaults to AUTOMATIC.
```

Settings for creating read replica clusters

For details about settings that you choose when you create a read replica cluster, see the following table. For more information about the AWS CLI options, see create-db-cluster. For more information about the Amazon Timestream for InfluxDB API parameters, see CreateDbCluster.

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Allocated storage	The amount of storage to allocate for each DB instance in your DB cluster (in gibibytes). For more informati	**CLI option:** `--allocated-storage` **API parameter:** `allocatedStorage`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
	on, see InfluxDB instance storage.	
Database port	The port number on which InfluxDB accepts connections. Valid Values: 1024-65535 Default: 8086 Constraints: The value can't be 2375-2376, 7788-7799, 8090, or 51678-51680.	**CLI option:** `--port` **API parameter:** `port`
DB cluster name	The name that uniquely identifies the DB cluster. DB instance names must be unique per customer and per region.	**CLI option:** `--name` **API parameter:** `name`
DB instance type	The compute and memory capacity of each DB instance in your Timestream for InfluxDB DB cluster, for example `db.influx.xlarge` . If possible, choose a DB instance class large enough that a typical query working set can be held in memory. When working sets are held in memory, the system can avoid writing to disk, which improves performance.	**CLI option:** `--db-instance-type` **API parameter:** `dbInstanceType`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
DB cluster parameter group	The ID of the DB parameter group to assign to your DB cluster. DB parameter groups specify how the database is configured. For example, DB parameter groups can specify the limit for query concurrency.	**CLI option:** `--db-parameter-group-identifier` **API parameter:** `dbParameterGroupIdentifier`
Deployment type	Specifies whether the DB cluster will be deployed as a multinode read replica or a Multi-AZ multinode read replica. Possible values: `MULTI_NODE_READ_REPLICAS`	**CLI option:** `--deployment-type` **API parameter:** `deploymentType`
VPC subnet ID	The DB subnet ID you want to use for the DB cluster. Select **Choose existing** to use an existing DB subnet group, then choose the required subnet group from the **Existing DB subnet groups** dropdown list. Choose **Automatic setup** to let Timestream for InfluxDB select a compatible DB subnet group.	**CLI option:** `--vpc-subnet-ids` **API parameter:** `vpcSubnetIds`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Organization	The name of the initial organization for the initial admin user in InfluxDB. An InfluxDB organization is a workspace for a group of users.	**CLI option:** `--organization` **API parameter:** `organization`
Bucket	The name of the initial InfluxDB bucket. All InfluxDB data is stored in a bucket. A bucket combines the concept of a database and a retention period (the duration of time that each data point persists). A bucket belongs to an organization.	**CLI option:** `--bucket` **API parameter:** `bucket`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Log exports	Configuration for sending InfluxDB engine logs to a specified S3 bucket. Configuration for S3 bucket log delivery: `s3Configuration -> (structure)` The name of the S3 bucket to deliver logs to: `bucketName -> (string)` Indicates whether log delivery to the S3 bucket is enabled: `enabled -> (boolean)` Shorthand syntax: `s3Configuration={bucketName=string,enabled=boolean}`	**CLI option:** `--log-delivery-configuration` **API parameter:** `logDeliveryConfiguration`
Password	The password of the initial admin user you created in InfluxDB. This password will allow you to access the InfluxDB UI to perform various administrative tasks and also use the InfluxDB CLI to create an operator token. These attributes will be stored in a secret created in AWS Secrets Manager in your account.	**CLI option:** `--password` **API parameter:** `password`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Username	The username of the initial admin user created in InfluxDB. Must start with a letter and can't end with a hyphen or contain two consecutive hyphens. For example, my-user1. This username will allow you to access the InfluxDB UI to perform various administr ative tasks and also use the InfluxDB CLI to create an operator token. These attributes will be stored in a secret created in AWS Secrets Manager in your account.	**CLI option:** `--username` **API parameter:** `username`
Public access	Indicates whether the DB cluster is accessible from outside the VPC. **Publicly accessible** gives the DB cluster a public IP address, meaning it's accessibl e outside the VPC. To be publicly accessible, the DB cluster also has to be in a public subnet in the VPC. **Not publicly accessibl e** makes the DB cluster accessible only from inside the VPC.	**CLI options:** `--publicly-accessible --no-publ icly-accessible` **API parameter:** `publiclyA ccessible`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
DB storage type	InfluxDB data. You can choose between three different types of provisioned Influx IOPS Included storage according to your workload's requirements. Possible values: • InfluxIOIncludedT1 • InfluxIOIncludedT2 • InfluxIOIncludedT3	**CLI options:** `--db-storage-type --no-publicly-accessible` **API parameter:** `dbStorageType`
VPC security group	A list of VPC security group IDs to associate with the DB instance.	**CLI options:** `--vpc-security-group-ids --no-publicly-accessible` **API parameter:** `vpcSecurityGroupIds`
VPC subnet IDs	A list of VPC subnet IDs to associate with the DB instance. Provide at least two VPC subnet IDs in different Availability Zones when deploying with a Timestream for InfluxDB DB cluster.	**CLI options:** `--vpc-subnet-ids` **API parameter:** `vpcSubnetIds`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Failover mode	How your cluster responds to a primary instance failure. You can configure this with the following options: `AUTOMATIC` : If the primary instance fails, the system automatically promotes a read replica to become the new primary instance. `NO_FAILOVER` : If the primary instance fails, the system attempts to restore the primary instance without promoting a read replica. The cluster remains unavailable until the primary instance is restored.	**CLI options:** `--failover-mode` **API parameter:** `failoverMode`

Important

As part of the DB cluster response object, you will receive an `influxAuthParametersSecretArn`. This will hold an ARN to a Secrets Manager secret in your account. It will only be populated after your InfluxDB DB instances are available. The secret contains Influx authentication parameters provided during the `CreateDbInstance` process. This is a **read-only** copy as any updates/modifications/deletions to this secret doesn't impact the created DB instance. If you delete this secret, our API response will still refer to the deleted secret ARN.

Connecting to a Timestream for InfluxDB read replica DB cluster

A Timestream for InfluxDB read replica DB cluster has two reachable DB instances instead of a single DB instance. Each connection is handled by a specific DB instance. When you connect to a read replica DB cluster, the hostname and port that you specify point to a fully qualified domain name called an *endpoint*.

The primary (writer) endpoint connects to the writer DB instance of the read replica DB cluster, which supports both read and write operations. The reader endpoint connects to the reader DB instance, which support only read operations.

Using endpoints, you can map each connection to the appropriate DB instance based on your use case. For example, to perform administrative or write statements, you can connect to whichever DB instance is the writer DB instance. To perform queries, you can connect to the reader endpoint. For diagnosis or tuning, you can connect to a specific DB instance endpoint, `/metrics`, to examine details about a specific DB instance.

For information about connecting to a DB instance, see Connecting to an Amazon Timestream for InfluxDB DB instance. For more information about connecting to read replica clusters, see the following topics.

Types of read replica cluster endpoints

An endpoint is represented by a unique identifier that contains a host address. Each Timestream for InfluxDB cluster has:

- A cluster endpoint.
- A cluster read-only endpoint.
- An instance endpoint for each instance in the cluster.

Cluster endpoint

A *cluster endpoint* (or *writer endpoint*) for a read replica cluster connects to the current writer DB instance for that DB cluster. This endpoint is the only one that can perform write operations such as:

- InfluxDB-specific administrative commands, e.g., creating, modifying, or deleting organizations, users, buckets, tasks, etc.
- Writing data to your database cluster.

You use the cluster endpoint for all write operations on the DB cluster, including writes, upserts, deletes, and all configuration and administrative changes.

In addition, you can use the cluster endpoint for read operations, such as queries.

If the current writer DB instance of a DB cluster fails, the read replica cluster automatically fails over to one of its replicas, promoting it as the new writer DB instance. During a failover, the DB cluster continues to serve connection requests to the cluster endpoint from the new writer DB instance, with minimal interruption of service. The read replica endpoint that was promoted to writer will stop serving reads until a new replica is deployed.

The following example illustrates a cluster endpoint for a read replica cluster:

```
ipvtdwa5se-wmyjrrjko.us-west-2.timestream-influxdb.amazonaws.com
```

Read-only endpoint

The *read-only endpoint* connects to any one of the read replica instances in the cluster. Read replicas will only support read operations, such as Flux or InfluxQL queries; in other words, all operations executed against the `/api/v2/query` endpoint for Flux queries or `/api/query` endpoint for InfluxQL v1-compatible queries. By processing those statements on the reader DB instances, this endpoint reduces the overhead on the writer DB instance. It also helps the cluster to handle a higher number of simultaneous queries.

The following example illustrates a reader endpoint for a read replica cluster. The read-only intent of a reader endpoint is denoted by the `-ro` within the cluster endpoint name.

```
ipvtdwa5se-wmyjrrjko-ro.us-west-2.timestream-influxdb.amazonaws.com
```

Instance endpoint

An *instance endpoint* connects to a specific DB instance within a read replica cluster. Each DB instance in a DB cluster has its own unique instance endpoint. Therefore, there is one instance endpoint for the current writer DB instance of the DB cluster (the primary), and there is one instance endpoint for each of the reader DB instances in the DB cluster.

The instance endpoint provides direct control over connections to the DB cluster. This control can help you address scenarios where using the cluster endpoint or reader endpoint might not be appropriate. For example, your client application might require more fine-grained load balancing

based on workload type. In this case, you can configure multiple clients to connect to different reader DB instances in a DB cluster to distribute read workloads.

The following example illustrates an instance endpoint for a DB instance in a read replica cluster:

```
mydbinstance-123456789012.us-east-1.timestream-influxdb.amazonaws.com
```

Modifying a read replica cluster for Amazon Timestream for InfluxDB

A read replica cluster has a writer DB instance and a reader DB instance in separate Availability Zones. Read replica clusters provide high availability, increased capacity for read workloads, and faster failover when compared to Multi-AZ deployments. For more information about read replica clusters, see Overview of Amazon Timestream for InfluxDB read replica clusters.

You can modify a read replica cluster to change its settings.

> **Important**
>
> You can't modify the DB instances within a read replica cluster. All modifications must be done at the DB cluster level.
> You can modify a read replica cluster using the AWS Management Console, the AWS CLI, or the Amazon Timestream for InfluxDB API.

Modify a read replica cluster for Amazon Timestream for InfluxDB

Using the AWS Management Console

To modify a read replica DB cluster using the console:

1. Sign in to the AWS Management Console and open the Amazon Timestream console.
2. In the navigation pane, choose **InfluxDB databases** and then choose the read replica cluster you want to modify.
3. Choose **Modify**. The **Modify DB cluster** page appears.
4. Choose any of the settings that you want. For information about each setting, see Settings for modifying read replica clusters.
5. After you have made your changes, choose **Continue** and check the summary of modifications.

6. On the confirmation page, review your changes. If they're correct, choose **Modify DB cluster** to save your changes. Alternatively, choose **Back** to edit your changes or **Cancel** to cancel your changes.

> **Important**
>
> Currently Amazon Timestream for InfluxDB only supports **Apply Immediately** updates for the read replica cluster. If you confirm the changes, your DB cluster will incur downtime while the changes are being applied.

Using the AWS CLI

To modify a DB instance using the AWS Command Line Interface, use the `update-db-cluster` command with the following parameters. Replace each *`user input placeholder`* with your own information.

```
aws timestream-influxdb update-db-cluster \
      --region region \
      --db-cluster-id db-cluster-id \
      --db-instance-type db.influx.4xlarge \
      --port 10000 \
      --failover mode NO_FAILOVER
```

Settings for modifying read replica clusters

For details about settings that you can use to modify a read replica cluster, see the following table. For more information about the AWS CLI options, see update-db-cluster.

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Database port	The port number on which InfluxDB accepts connections. Valid Values: 1024-65535	**CLI option:** `--port` **API parameter:** `port`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
	Default: 8086 Constraints: The value can't be 2375-2376, 7788-7799, 8090, or 51678-51680.	
DB instance type	The compute and memory capacity of each DB instance in your Timestream for InfluxDB DB cluster, for example `db.influx.xlarge` . If possible, choose a DB instance class large enough that a typical query working set can be held in memory. When working sets are held in memory, the system can avoid writing to disk, which improves performance.	**CLI option:** `--db-instance-type` **API parameter:** `dbInstanceType`
DB cluster parameter group	The ID of the DB parameter group to assign to your DB cluster. DB parameter groups specify how the database is configured. For example, DB parameter groups can specify the limit for query concurrency.	**CLI option:** `--db-parameter-group-identifier` **API parameter:** `dbParameterGroupIdentifier`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Log exports	Configuration for sending InfluxDB engine logs to a specified S3 bucket. Configuration for S3 bucket log delivery: `s3Configuration -> (structure)` The name of the S3 bucket to deliver logs to: `bucketName -> (string)` Indicate whether log delivery to the S3 bucket is enabled: `enabled -> (boolean)` Shorthand syntax: `s3Configuration={bucketName=string, enabled=boolean}`	**CLI option:** `--log-delivery-configuration` **API parameter:** `logDeliveryConfiguration`

Console setting	Setting description	CLI option and Timestream for InfluxDB API parameter
Failover mode	Configure how your cluster responds to a primary instance failure using the following options: `AUTOMATIC` : If the primary instance fails, the system automatically promotes a read replica to become the new primary instance. `NO_FAILOVER` : If the primary instance fails, the system attempts to restore the primary instance without promoting a read replica. The cluster remains unavailable until the primary instance is restored.	**CLI option:** `--failover-mode` **API parameter:** `failoverMode`

Rebooting a read replica cluster in Amazon Timestream for InfluxDB

You can reboot a read replica cluster in the event of any health issues.

Rebooting a read replica cluster for Amazon Timestream for InfluxDB

Using the AWS Management Console

To reboot a read replica DB cluster using the console:

1. Sign in to the AWS Management Console and open the Amazon Timestream console.
2. In the navigation pane, choose **InfluxDB databases** and then choose the read replica cluster you want to reboot.
3. Choose **Restart database**.

4. Choose **Confirm and Restart**.

Using the AWS CLI

To reboot a DB instance using the AWS Command Line Interface, use the `reboot-db-cluster` command with the following parameters. Replace each *`user input placeholder`* with your own information.

```
aws timestream-influxdb reboot-db-cluster \
      --region region \
      --db-cluster-id db-cluster-id \
```

Creating CloudWatch alarms to monitor Amazon Timestream for InfluxDB

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period that you specify. The alarm can also perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon SNS topic or Amazon EC2 Auto Scaling policy.

Alarms invoke actions for sustained state changes only. CloudWatch alarms don't invoke actions simply because they are in a particular state. The state must have changed and have been maintained for a specified number of time periods.

You can set CloudWatch alarms on any of the available metrics for Timestream for InfluxDB, including `CPUUtilization`, `MemoryUtilization`, `DiskUtilization`, and `ReplicaLag`.

We recommend to start creating `DiskUtilization`-related alarms for your Timestream for InfluxDB databases, since out-of-storage space issues can turn out to be fairly problematic to InfluxDB. We recommend setting alerts to be sent whenever `DiskUtilization` goes over approximately 75–80 percent.

To set an alarm using the AWS CLI

Call `put-metric-alarm`. For more information, see put-metric-alarm in the *AWS CLI Command Reference*.

To set an alarm using the CloudWatch API

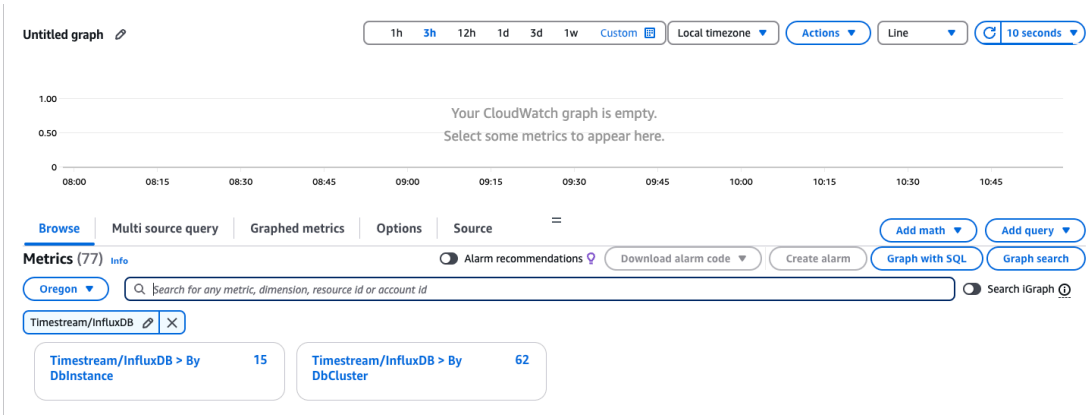
Call `PutMetricAlarm`. For more information, see PutMetricAlarm in the *Amazon CloudWatch API Reference*. For more information about setting up Amazon SNS topics and creating alarms, see Using Amazon CloudWatch alarms.

Tutorial: Create an Amazon CloudWatch alarm for Multi-AZ cluster replica lag for Amazon Timestream for InfluxDB

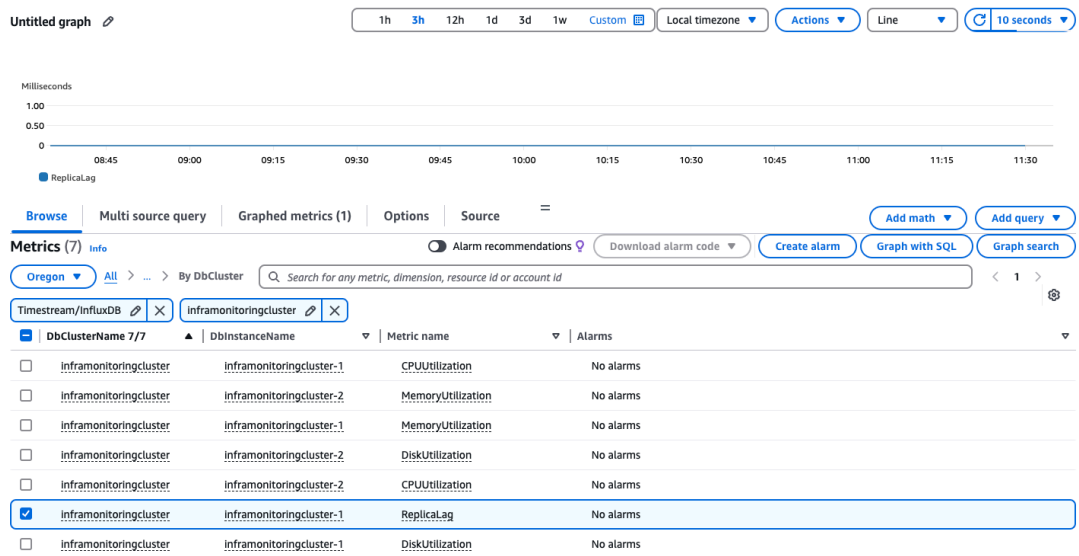
You can create an Amazon CloudWatch alarm that sends an Amazon SNS message when replica lag for a Multi-AZ DB cluster has exceeded a threshold. An alarm watches the `ReplicaLag` metric over a time period that you specify. The action is a notification sent to an Amazon SNS topic or Amazon EC2 Auto Scaling policy.

To set a CloudWatch alarm for Multi-AZ DB cluster replica lag

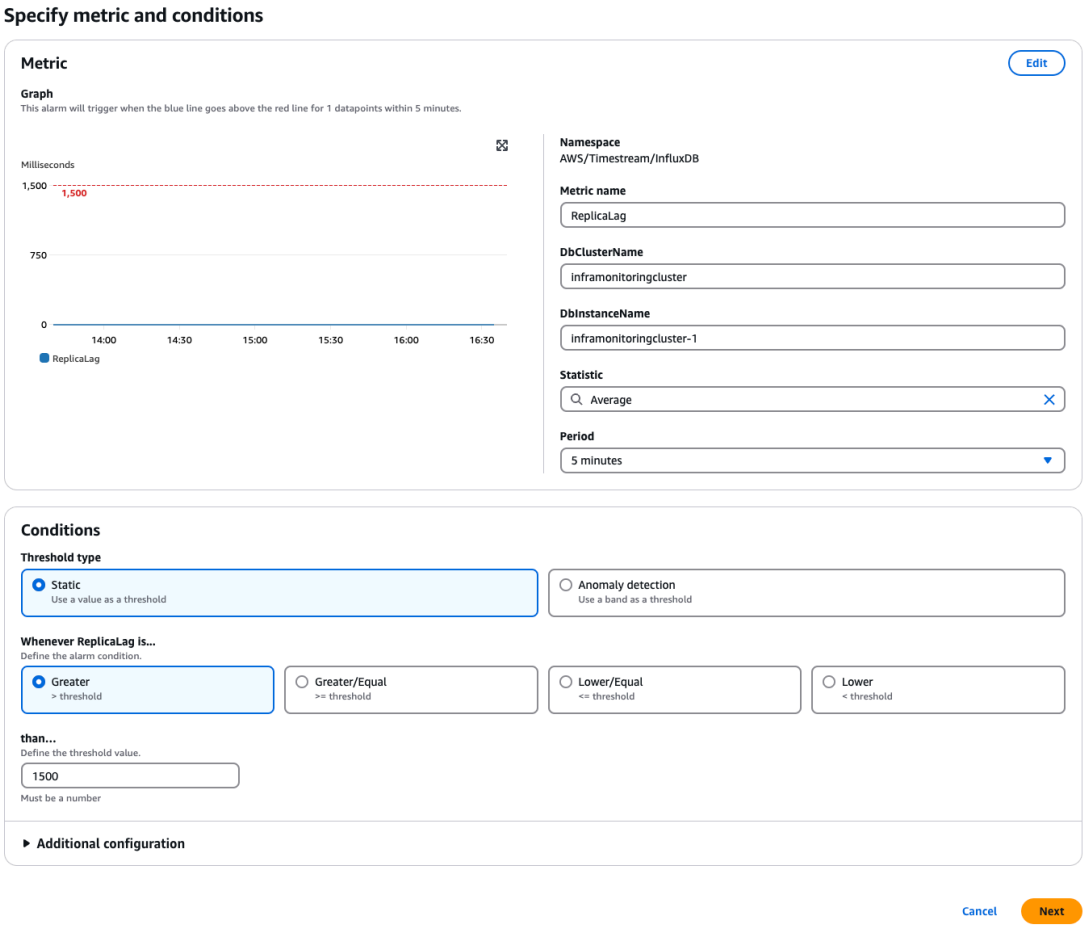
1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Alarms**, then **All alarms**.
3. Choose **Create alarm**.
4. On the **Specify metric and conditions** page, choose **Select metric**.
5. In the search box, enter the name of your DB cluster, select **Timestream/InfluxDB**, **By DbCluster**, and then select your cluster.



6. The following image shows the **Select metric** page with a read replica cluster named `inframonitoringcluster` selected. Choose the metric you want to create an alarm for, in this case `ReplicaLag`. Click **Select metric**.



7. On the **Specify metric and conditions** page, customize the following fields:



a. Select a period of time for your calculations in the **Period** section.

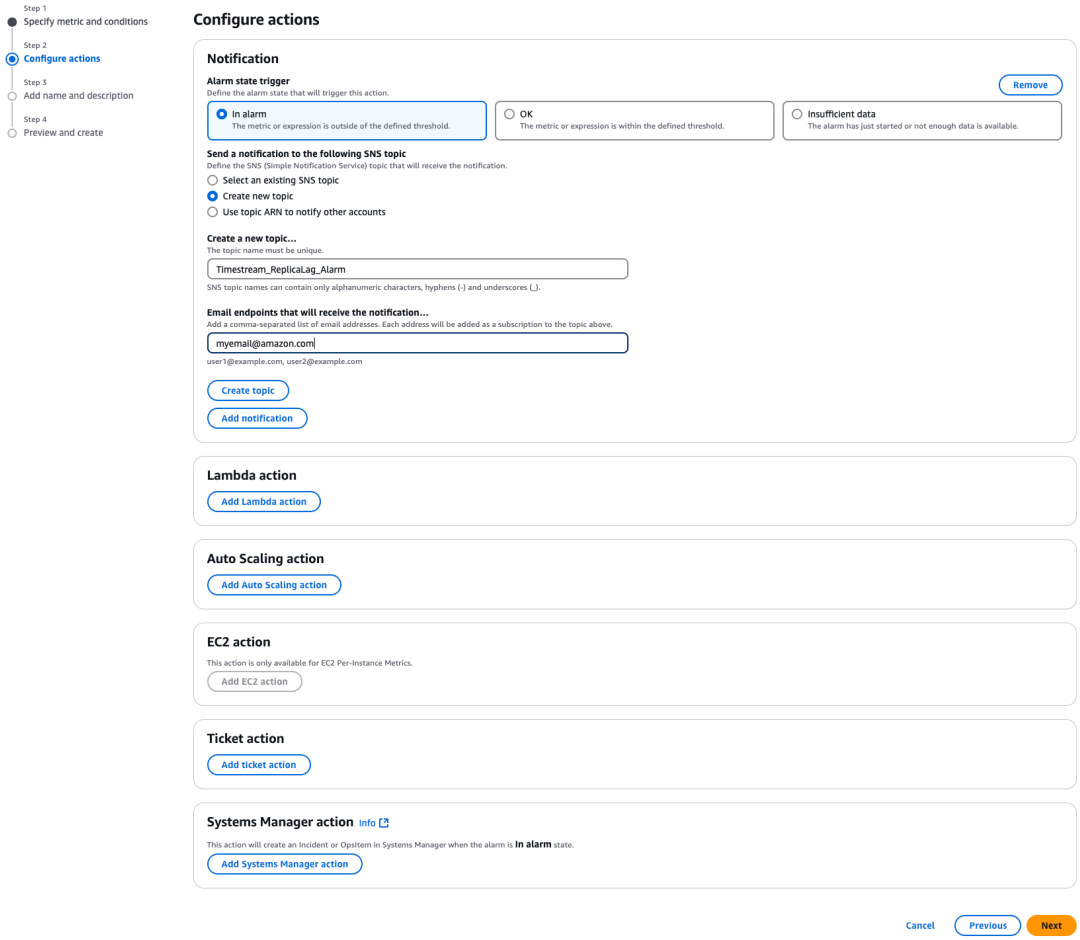
b. Set up the conditions related to your alarm. For **Threshold type**, you can choose between **Static** and **Anomaly detection**.

   In this case, we will use **Static** since we know how our workload behaves. Each workload might have different requirements when it comes to what is considered "healthy."

c. Select your threshold value. In the case of **Static** threshold values, these will be in milliseconds.

d. Choose **Next**.

8. On the **Configure actions** page, in the **Notification** section, customize the following settings:



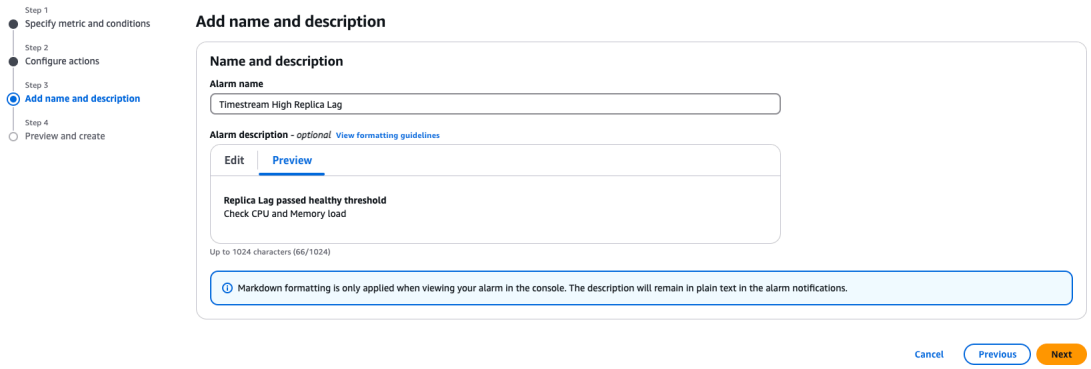
a. For **Alarm state trigger**, select **In alarm**.

b. Choose **Create new topic** in **Send a notification to the following SNS topic**.

c. Enter a unique topic name and a valid email address that will receive the notification.

d. Choose **Create topic**. Scroll down and choose **Next**.

9. On the **Add name and description** page, enter an **Alarm name** and **Alarm description**. Choose **Next**.



10. Review your alarm settings on the **Preview and create** page, and then choose **Create alarm**.

> **Important**
>
> To keep your Timestream for InfluxDB cluster in a healthy state, we also recommend monitoring and creating alarms for `CPUUtilization` and `MemoryUtilization` that consistently exceed a healthy 85 percent usage and `DiskUtilization` that exceeds 75 percent.

Read replica licensing through AWS Marketplace

To use Timestream for InfluxDB read replicas, you will need to activate the Timestream for InfluxDB read replicas add-on license through AWS Marketplace. Once the license is active, you will pay an hourly rate to use read replica clusters. You will only pay for the hours your read replica cluster is active. If you subscribe to the license but have no active Timestream for InfluxDB read replica clusters, you will not be charged.

Topics

- Read replica licensing terminology
- Payments and billing
- Subscribing to the InfluxDB read replica add-on on Marketplace listings

Read replica licensing terminology

This page uses the following terminology when discussing the Amazon Timestream for InfluxDB integration with AWS Marketplace.

SaaS subscription

In AWS Marketplace, software-as-a-service (SaaS) products such as the pay-as-you-go license model adopt a usage-based subscription model. InfluxData, the software seller for the read replica add-on, tracks your usage and you pay only for what you use.

InfluxData Marketplace fees

Fees charged for the InfluxDB read replica add-on software license usage by InfluxData. These service fees are metered through AWS Marketplace and appear on your AWS bill under the AWS Marketplace section.

Amazon Timestream for InfluxDB fees

Fees that AWS charges for the Amazon Timestream for InfluxDB services, which excludes licenses when using Timestream for InfluxDB read replica clusters. Fees are metered through the Amazon Timestream for InfluxDB service being used and appear on your AWS bill.

Payments and billing

Timestream for InfluxDB integrates with AWS Marketplace to offer hourly, pay-as-you-go licenses for the read replica add-on. The read replica Marketplace fees cover the license costs of the read replica add-on software, and the Amazon Timestream fees cover the costs of your Timestream for InfluxDB read replica cluster usage. For information about pricing, see Amazon Timestream pricing.

To stop these fees, you must delete any Timestream for InfluxDB read replica clusters. In addition, you can remove your subscriptions to AWS Marketplace for read replica add-on license. If you remove your subscriptions without deleting your read replica clusters, Amazon Timestream will continue to bill you for the use of the read replica clusters. For more information, see Considerations when deleting replicas.

You can view bills and manage payments for your Timestream for InfluxDB read replica cluster in the AWS Billing console. Your bills includes two charges: one for your usage of InfluxData's licensed add-on through AWS Marketplace, and one for your usage of Amazon Timestream. For more information about billing, see Understanding your bill in the *AWS Billing and Cost Management User Guide*.

Subscribing to the InfluxDB read replica add-on on Marketplace listings

To use the read replica add-on license through AWS Marketplace, you must use the Amazon Timestream AWS Management Console to subscribe to the InfluxDB read replica add-on. You cannot complete these tasks through the AWS CLI or the Timestream for InfluxDB API.

Topics

- Subscribe from Amazon Timestream AWS Management Console
- Subscribe to the InfluxDB read replica add-on in AWS Marketplace

> **Note**
>
> If you want to create your read replica cluster by using the AWS CLI or the Timestream for InfluxDB API, you must complete this step first.

Subscribe from Amazon Timestream AWS Management Console

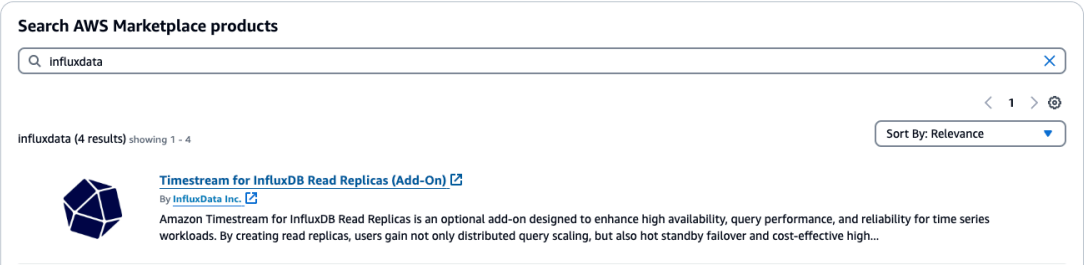
You can subscribe to the InfluxDB read replica add-on using the Timestream Management Console. Start the **Create InfluxDB Database** flow and follow the steps. For more information, see Creating a Timestream for InfluxDB read replica cluster.

Subscribe to the InfluxDB read replica add-on in AWS Marketplace

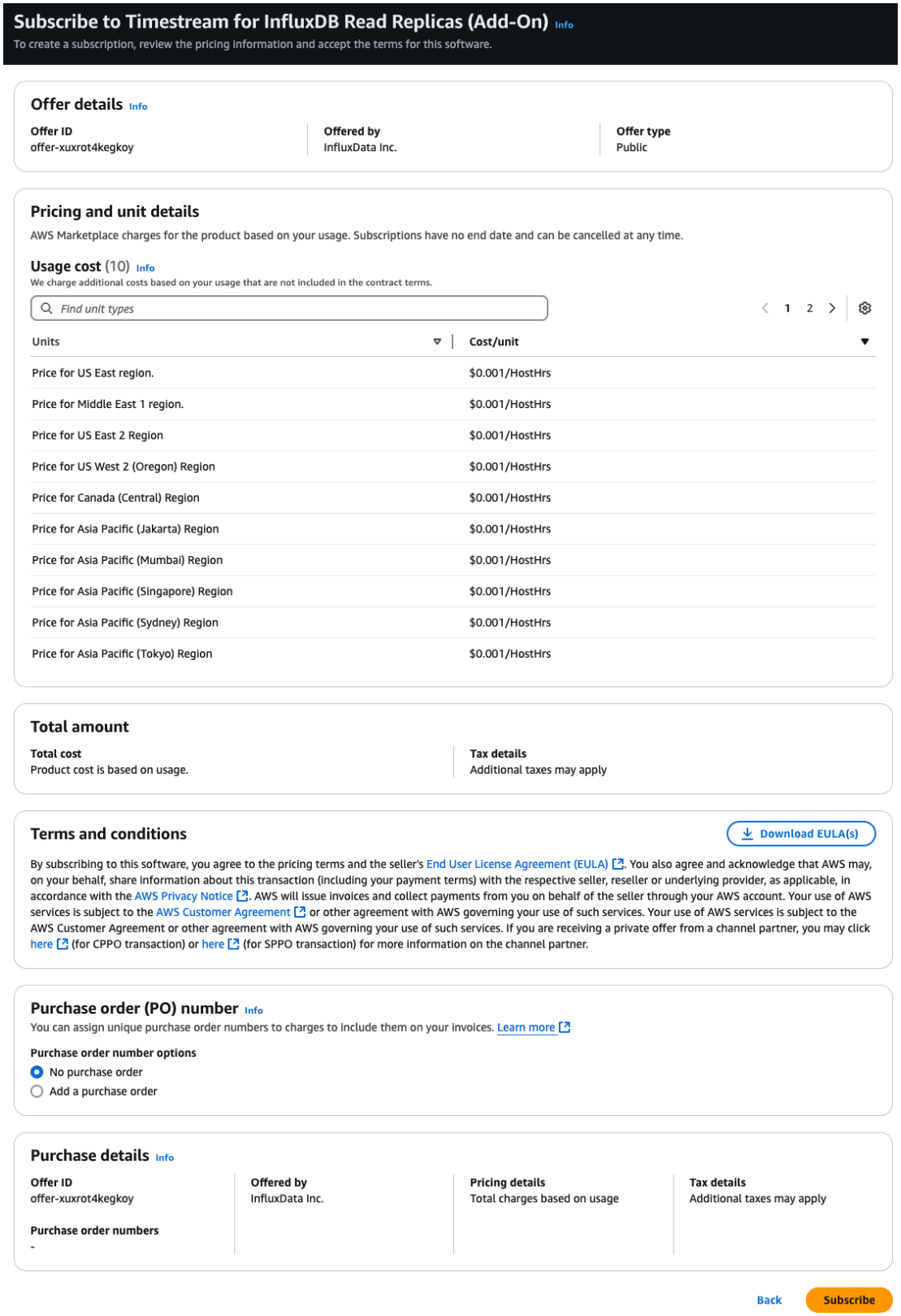
To use the InfluxDB add-on license with AWS Marketplace, you need to have an active AWS Marketplace subscription for the InfluxDB read replica add-on. You will need to subscribe to a single add-on offer and that will allow you to create any instance type you need in any of the available regions. For information about AWS Marketplace subscriptions, see SaaS products through AWS Marketplace in the *AWS Marketplace Buyer Guide*.

We recommend that you subscribe to InfluxDB in AWS Marketplace *before* you start creating a DB instance.

1. Navigate to the AWS Marketplace and search for InfluxData.



2. Select **Timestream for InfluxDB Read Replicas (Add-On)**.
3. Select **View purchase options**.
4. Review the End User License Agreement and choose **Subscribe**.



5. You can now create your Timestream for InfluxDB read replica cluster using the Timestream Management Console, CLI, or API.

Managing DB instances

This section covers various aspects of managing Amazon Timestream for InfluxDB instance to ensure optimal performance, availability, and monitoring capabilities. It provides guidance on updating configurations of your database instances, handling multi-AZ deployments, and failover processes. It also explains how to delete database instances and set up log viewing for your InfluxDB instances.

Topics

- Updating DB instances
- Maintenance windows
- Deleting a DB instance
- Rebooting a DB instance
- Multi-AZ DB instance deployments
- Setup to view InfluxDB logs on Timestream Influxdb Instances
- Monitoring and Configuration Optimization for Timestream for InfluxDB 2

Updating DB instances

You can update the following configuration parameters of your Timestream for InfluxDB instance:

- Instance class
- Storage type
- Allocated storage (increase only)
- Deployment type
- Parameter group
- Log delivery configuration

> **Important**
>
> We recommend you test all changes on a test instance before modifying the production instance to understand their impact, especially when upgrading database versions. Review the impact on your database and applications before updating settings. Some modifications require a DB instance reboot, resulting in downtime.

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**, and then choose the DB instance that you want to modify.
3. Choose **Modify**.
4. On the **Modify DB instance** page, make the desired changes.
5. Choose **Continue** and check the summary of modifications.
6. Choose **Next**.
7. Review your changes.
8. Choose **Modify instance** to apply your changes.

> **Note**
>
> These modifications require a reboot of the Influx DB instance and can cause an outage in some cases.

Using the AWS Command Line Interface

To update a DB instance by using the AWS Command Line Interface, call the `update-db-instance` command. Specify the DB instance identifier and the values for the options that you want to modify. For information about each option, see Settings for DB instances.

Example

The following code modifies *`my-db-instance`* by setting a different `db-parameter-group-name`. Replace each *`user input placeholder`* with your own information. The changes are applied immediately.

For Linux, macOS, or Unix:

```
aws timestream-influxdb update-db-instance \
    --identifier my-db-instance \
    --db-storage-type desired-storage-type \
    --allocated-storage desired-allocated-storage \
    --db-instance-type desired-instance-type \
```

```
    --deployment-type desired-deployment-type \
    --db-parameter-group-name new-param-group \
    --port 8086
```

For Windows:

```
aws timestream-influxdb update-db-instance ^
    --identifier my-db-instance ^
    --db-storage-type desired-storage-type ^
    --allocated-storage desired-allocated-storage ^
    --db-instance-type desired-instance-type ^
    --deployment-type desired-deployment-type ^
    --db-parameter-group-name new-param-group
    --port 8086
```

Maintenance windows

Periodically, Amazon Timestream for InfluxDB performs maintenance on Amazon Timestream for InfluxDB resources. Maintenance most often involves updates to the following resources in your DB instance:

- Underlying hardware
- Underlying operating system (OS)
- Database engine version

Updates to the operating system most often occur for security issues.

Some maintenance items require that Amazon Timestream for InfluxDB take your DB instance offline for a short time. Maintenance items that require a resource to be offline include required operating system or database patching. Required patching is automatically scheduled only for patches that are related to security and instance reliability. Such patching occurs infrequently, typically once every few months. It seldom requires more than a fraction of your maintenance window.

Maintenance window

Every Amazon Timestream for InfluxDB DB instance has a weekly maintenance window during which maintenance is performed. You can configure your maintenance window in two ways:

- **Service managed (default)**: Amazon Timestream for InfluxDB determines the optimal maintenance window for your resource.
- **Customer managed**: You specify a preferred maintenance window using the format `ddd:HH:MM-ddd:HH:MM` (for example, `Sun:02:00-Sun:04:00`). The window must be at least 2 hours and no more than 24 hours. Cross-midnight windows are supported.

You can set your preferred maintenance window when creating a DB instance or change it later using the `update-db-instance` API.

Timezone

You can specify a timezone for your maintenance window using the `timezone` field. When set, window times are interpreted in the specified timezone. The `timezone` field is required. Use IANA timezone identifiers such as `America/New_York` or `Asia/Tokyo`. The system handles Daylight Saving Time transitions automatically.

CLI examples

Create a DB instance with a custom maintenance window:

```
aws timestream-influxdb create-db-instance \
  --name "my-influxdb" \
  --db-instance-type db.influx.medium \
  --allocated-storage 50 \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --maintenance-schedule '{
    "timezone": "America/New_York",
    "preferredMaintenanceWindow": "Sun:02:00-Sun:04:00"
  }' \
  --region us-west-2
```

Update the maintenance window on an existing DB instance:

```
aws timestream-influxdb update-db-instance \
  --identifier <instance-identifier> \
  --maintenance-schedule '{
    "timezone": "Asia/Tokyo",
    "preferredMaintenanceWindow": "Wed:03:00-Wed:06:00"
  }' \
```

```
  --region us-west-2
```

Revert to service managed:

```
aws timestream-influxdb update-db-instance \
  --identifier <instance-identifier> \
  --maintenance-schedule '{
    "timezone": "UTC",
    "preferredMaintenanceWindow": ""
  }' \
  --region us-west-2
```

> **Important**
>
> If a required maintenance action has been deferred for more than 25 days, the service may apply maintenance outside of your preferred window to ensure the security and reliability of your resource.

During maintenance, the DB instance status changes to `MAINTENANCE`. After completion, the status returns to `AVAILABLE`.

Supported timezones

Use IANA timezone identifiers. Timezone abbreviations such as `EST`, `PST`, and `GMT+5` are not supported.

Timezone	Description
`UTC`	Coordinated Universal Time (default)
`America/New_York`	US Eastern
`America/Chicago`	US Central
`America/Denver`	US Mountain
`America/Los_Angeles`	US Pacific
`America/Sao_Paulo`	Brazil

Timezone	Description
`Europe/London`	UK
`Europe/Paris`	Central Europe
`Europe/Berlin`	Germany
`Asia/Tokyo`	Japan
`Asia/Shanghai`	China
`Asia/Singapore`	Singapore
`Asia/Mumbai`	India
`Asia/Dubai`	UAE
`Australia/Sydney`	Australia Eastern
`Pacific/Auckland`	New Zealand

Considerations

- Maintenance windows define when maintenance *can* occur, not when it *will* occur. Maintenance is performed as needed, typically no more than once per week.
- Maintenance is required at least once per month for security and reliability patching.
- For Multi-AZ deployments, maintenance is performed on the standby first, then a failover occurs, minimizing downtime.
- If you use a timezone with DST transitions, avoid scheduling maintenance between 1:00 AM and 3:00 AM to prevent skipped windows during spring forward.

Deleting a DB instance

Deleting a DB instance has an effect on instance recoverability, and snapshot availability. Consider the following issues:

- If you want to delete all Timestream for InfluxDB resources, note that the DB instances resources incur billing charges.
- When the status for a DB instance is deleting, its CA certificate value doesn't appear in the Timestream for InfluxDB console or in output for AWS Command Line Interface commands or Timestream API operations.
- The time required to delete a DB instance varies depending on how much data is deleted, and whether a final snapshot is taken.

You can delete a DB instance using the AWS Management Console, the AWS Command Line Interface, or the Timestream API. You must provide the name of the DB instance:

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**, and then choose the DB instance that you want to delete.
3. Choose **Delete**.
4. Enter *confirm* in the box.
5. Choose **Delete**.

Using the AWS Command Line Interface

To find the instance IDs of the DB instances in your account, call the `list-db-instances` command:

```
aws timestream-influxdb list-db-instances \
--endpoint-url YOUR_ENDPOINT \
--region YOUR_REGION
```

To delete a DB instance by using the AWS CLI, call the `delete-db-instance` command with the following options:

```
aws timestream-influxdb list-db-instances \
--identifier YOUR_DB_INSTANCE \
```

Example

For Linux, macOS, or Unix:

```
aws timestream-influxdb delete-db-instance \
    --identifier mydbinstance
```

For Windows:

```
aws timestream-influxdb delete-db-instance ^
    --identifier mydbinstance
```

Rebooting a DB instance

You can reboot a DB instance using the AWS Management Console, the AWS Command Line Interface, or the Timestream API. You must provide the ID of the DB instance:

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**, and then choose the DB instance that you want to reboot.
3. Choose **Restart database**.
4. Choose **Confirm and Restart**.

Using the AWS Command Line Interface

To reboot a DB instance by using the AWS CLI, call the `reboot-db-instance` command with the following options:

Example Commands

For Linux, macOS, or Unix:

```
aws timestream-influxdb reboot-db-instance \
    --region YOUR_REGION \
```

```
    --identifier YOUR_INSTANCE_ID
```

For Windows:

```
aws timestream-influxdb reboot-db-instance ^
    --region YOUR_REGION ^
    --identifier YOUR_INSTANCE_ID
```

Multi-AZ DB instance deployments

Amazon Timestream for InfluxDB provides high availability and failover support for DB instances using Multi-AZ deployments with a single standby DB instance. This type of deployment is called a Multi-AZ DB instance deployment. Amazon Timestream for InfluxDB use the Amazon failover technology.

In a Multi-AZ DB instance deployment, Amazon Timestream automatically provisions and maintains a synchronous standby replica in a different Availability Zone. The primary DB instance is synchronously replicated across Availability Zones to a standby replica to provide data redundancy. Running a DB instance with high availability can enhance availability during DB instance failure and Availability Zone disruption. For more information on , see AWS Regions and Availability Zones.

> **Note**
>
> The high availability option isn't a scaling solution for read-only scenarios. You can't use a standby replica to serve read traffic.

Using the Amazon Timestream console, you can create a Multi-AZ DB instance deployment by simply specifying **Create a standby instance** option in the **Availability and durability configuration** section when creating a DB instance. You can also specify a Multi-AZ DB instance deployment with the AWS Command Line Interface or Amazon Timestream API. Use the `create-db-instance` or CLI command, or the `CreateDBInstance` API operation.

DB instances using Multi-AZ DB instance deployments can have increased write and commit latency compared to a Single-AZ deployment. This can happen because of the synchronous data replication that occurs. You might have a change in latency if your deployment fails over to the standby replica, although AWS is engineered with low-latency network connectivity between . For production workloads, we recommend that you use IOPS Included storage 12K or 16K IOPS for

fast, consistent performance. For more information about DB instance classes, see DB instance classes.

Configuring and managing a multi-AZ deployment

Timestream for InfluxDB Multi-AZ deployments can only have one standby. When the deployment has one standby DB instance, it's called a Multi-AZ DB instance deployment. A Multi-AZ DB instance deployment has one standby DB instance that provides failover support, but doesn't serve read traffic.

> **Important**
>
> Your instance must have at least two subnets associated with it to execute Single-AZ to Multi-AZ updates. Once the instance is created, you can't modify its deployment mode from Single-AZ to Multi-AZ .

You can use the AWS Management Console to determine whether your DB instance is a Single-AZ or Multi-AZ deployment.

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB databases**, and then choose **DB identifier**.

A Multi-AZ DB instance deployment has the following characteristics:

- There is only one row for the DB instance.
- The value of Role is Instance or Primary.
- The value of Multi-AZ is Yes.

Failover process for Amazon Timestream

If a planned or unplanned outage of your DB instance results from an infrastructure defect, Amazon Timestream for InfluxDB automatically switches to a standby replica in another Availability Zone if you have turned on Multi-AZ. The time that it takes for the failover to complete depends on

the database activity and other conditions at the time the primary DB instance became unavailable. Failover times are typically 60–120 seconds. However, large transactions or a lengthy recovery process can increase failover time. When the failover is complete, it can take additional time for the Timestream console to reflect the new Availability Zone.

> **Note**
>
> Amazon Timestream handles failovers automatically so you can resume database operations as quickly as possible without administrative intervention. The primary DB instance switches over automatically to the standby replica if any of the conditions described in the following table occurs.

Failover reason	Description
The operating system underlying the Timestream database instance is being patched in an offline operation.	A failover was triggered during the maintenan ce window for an OS patch or a security update.
The primary host of the Timestream Multi-AZ instance is unhealthy.	The Multi-AZ DB instance deployment detected an impaired primary DB instance and failed over.
The primary host of the Timestream Multi-AZ instance is unreachable due to loss of network connectivity.	Timestream monitoring detected a network reachability failure to the primary DB instance and triggered a failover.
The Timestream instance was modified by customer.	An Timesteam for InfluxDB DB instance modification triggered a failover. For more information, see Updating DB instances.
The Timestream Multi-AZ primary instance is busy and unresponsive.	The primary DB instance is unresponsive. We recommend that you do the following : * Examine the event for excessive CPU, memory, or swap space usage. * Evaluate your workload to determine whether you're using the appropriate DB instance class. For more information, see DB instance classes.

Failover reason	Description
The storage volume underlying the primary host of the Timestream Multi-AZ instance experienced a failure.	The Multi-AZ DB instance deployment detected a storage issue on the primary DB instance and failed over.

Setting the JVM TTL for DNS name lookups

The failover mechanism automatically changes the Domain Name System (DNS) record of the DB instance to point to the standby DB instance. As a result, you need to re-establish any existing connections to your DB instance. In a Java virtual machine (JVM) environment, due to how the Java DNS caching mechanism works, you might need to reconfigure JVM settings.

The JVM caches DNS name lookups. When the JVM resolves a host name to an IP address, it caches the IP address for a specified period of time, known as the *time-to-live* (TTL).

Because AWS resources use DNS name entries that occasionally change, we recommend that you configure your JVM with a TTL value of no more than 60 seconds. Doing this makes sure that when a resource's IP address changes, your application can receive and use the resource's new IP address by requerying the DNS.

On some Java configurations, the JVM default TTL is set so that it never refreshes DNS entries until the JVM is restarted. Thus, if the IP address for an AWS resource changes while your application is still running, it can't use that resource until you manually restart the JVM and the cached IP information is refreshed. In this case, it's crucial to set the JVM's TTL so that it periodically refreshes its cached IP information.

You can get the JVM default TTL by retrieving the `networkaddress.cache.ttl` property value:

```
String ttl = java.security.Security.getProperty("networkaddress.cache.ttl");
```

> **Note**
>
> The default TTL can vary according to the version of your JVM and whether a security manager is installed. Many JVMs provide a default TTL less than 60 seconds. If you're using such a JVM and not using a security manager, you can ignore the rest of this topic.
> To modify the JVM's TTL, set the networkaddress.cache.ttl property value. Use one of the following methods, depending on your needs:

- To set the property value globally for all applications that use the JVM, set `networkaddress.cache.ttl` in the `$JAVA_HOME/jre/lib/security/java.security` file.

```
networkaddress.cache.ttl=60
```

- To set the property locally for your application only, set `networkaddress.cache.ttl` in your application's initialization code before any network connections are established.

```
java.security.Security.setProperty("networkaddress.cache.ttl" , "60");
```

Setup to view InfluxDB logs on Timestream Influxdb Instances

By default InfluxDB generates logs that go to stdout. For more information, see Manage InfluxDB logs

To view InfluxDB logs generated from the Instance you have created through Timestream InfluxDB, we provide the opportunity to provide hourly logs. These logs will go to a specified S3 bucket that you must create before creating your instance.

- Before creating the instance, the provided Amazon S3 bucket must also give Timestream-InfluxDB permission to send logs to this bucket by providing a bucket policy with Timestream InfluxDB Service Principal as following (replace *{BUCKET_NAME}* with the actual name of your Amazon S3 bucket:

  JSON

```
{
    "Version":"2012-10-17",
    "Id": "PolicyForInfluxLogs",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "timestream-influxdb.amazonaws.com"
            },
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::{BUCKET_NAME}/InfluxLogs/*"
        }
```

```
    ]
}
```

- The bucket provided must be in the same account and same Region of your created Timestream InfluxDB instance

  Here is the command you can call to make an instance to receive influx logs:

```
aws timestream-influxdb create-db-instance \
    --name myinfluxDbinstance \
    --allocated-storage 400 \
    --db-instance-type db.influx.4xlarge \
    --vpc-subnet-ids subnetid1 subnetid2
    --vpc-security-group-ids mysecuritygroup \
    --username masterawsuser \
    --password \
    --db-storage-type InfluxIOIncludedT2
```

  Here is the format of this parameter.

```
-- log-delivery-configuration
{
    "S3Configuration": {
      "BucketName": "string",
      "Enabled": true|false
    }
}
```

- This field is not required and logging is not enabled by default.
- Not setting this field is the same as not having logs enabled.
- Logs will be sent to specified bucket with a prefix of `InfluxLogs/`.
- After creating the instance, you can modify the log delivery configuration with the `update-db-instance` API command.

InfluxDB offers different types of logs. These can be configured by setting the InfluxDB Parameters. Use the flux-log-enabled and log-level parameters to configure the type of logs that is emitted from the instance. For more information, see Supported parameters and parameter values.

Monitoring and Configuration Optimization for Timestream for InfluxDB 2

Overview

Effective monitoring and configuration optimization are critical for maintaining optimal performance, reliability, and cost-efficiency in your Timestream for InfluxDB deployment. This guide provides comprehensive guidance on CloudWatch metrics, performance thresholds, and configuration tuning strategies to help you proactively manage your InfluxDB instances.

CloudWatch Metrics Reference

Amazon CloudWatch provides detailed metrics for monitoring your Timestream for InfluxDB instances. Understanding these metrics and their thresholds is essential for maintaining system health and performance.

Resource Utilization Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds
CPUUtilization	DbInstanceName	Percentage of CPU being used	Perce	• Development/Growing: < 70% • Production: < 80% • Critical Alert: > 90% for 5+ min
MemoryUtilization	DbInstanceName	Percentage of memory being used	Perce	• Development/Growing: < 70% • Production: < 80% • Critical Alert: > 90%
HeapMemoryUsage	DbInstanceName	Amount of heap memory in use	Bytes	• Monitor for steady growth or spikes • Alert: Approaching max heap size

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds
ActiveMemoryAllocation	DbInstanceName	Current active memory allocation	Bytes	• Monitor for unexpected spikes • Compare against total available memory
DiskUtilization	DbInstanceName	Percentage of disk space being used	Perce	• Development/Growing: < 70% • Production: < 75% • Critical Alert: > 85%

I/O Operations Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds
ReadOpsPerSec	DbInstanceName	Number of read operations per second	Count Second	Maintain ≥ 30% headroom below provisioned IOPS Example: 12K IOPS → keep < 8,400 IOPS total
WriteOpsPerSec	DbInstanceName	Number of write operations per second	Count Second	Maintain ≥ 30% headroom below provisioned IOPS Example: 12K IOPS → keep < 8,400 IOPS total

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds	
TotalIOpsPerSec	DbInstanceName	Total I/O operations per second (read + write)	Count/Second	Maintain ≥ 30% headroom below provisioned IOPS Monitor against instance class capabilities	

Throughput Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds	
ReadThroughput	DbInstanceName	Data read throughput	Bytes/Second	Monitor against storage throughput limits	
WriteThroughput	DbInstanceName	Data write throughput	Bytes/Second	Monitor against storage throughput limits	

API Performance Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds	
APIRequestRate	DbInstanceName, Endpoint, Status	Rate of API requests to specific endpoints with status codes (2xx, 4xx, 5xx)	Count/Second	Error rates: • 4xx errors: < 1% of requests • 5xx errors: < 0.1% of requests	

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds
				• Alert: Sudden spikes in error rates
QueryResponseVolume	DbInstanceName, Endpoint, Status	Volume of query responses by endpoint and status code	Bytes	• Monitor for unusually large responses • Alert: Responses > 10MB consistently

Query Execution Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds
QueryRequestsTotal	DbInstanceName, Result	Total count of query requests by result type (success, runtime_error, compile_error, queue_error)	Count	Success rate: > 99% Error rates: • runtime_error: < 0.5% • compile_error: < 0.1% • queue_error: < 0.1%

Data Organization Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Critical Thresholds
SeriesCardinality	DbInstanceName, Bucket	Number of unique time	Count	• < 100K: Excellent performance

CloudWatch Metric Name	Dimensions	Description	Unit	Critical Thresholds
		series in a bucket		• < 1M: Good performance • 1M - 5M: Moderate impact, requires tuning • 5M - 10M: Significant impact, careful optimization required • > 10M: CRITICAL - Consider InfluxDB 3.0
TotalBuckets	DbInstanceName	Total number of buckets in the instance	Coun	• Monitor growth over time • Consider consolidation if > 100 buckets

System Health Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds
EngineUptime	DbInstanceName	Time the InfluxDB engine has been running	Secor	Monitor for unexpected restarts Alert: Uptime resets unexpectedly
WriteTimeouts	DbInstanceName	Number of write operations that timed out	Coun	Alert: > 0.1% of write operations

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds	
				Critical: Increasing trend	

Task Management Metrics

CloudWatch Metric Name	Dimensions	Description	Unit	Recommended Thresholds	
ActiveTaskWorkers	DbInstanceName	Number of active task workers	Coun	Monitor against configured task worker limit Alert: Consistently at maximum	
TaskExecutionFailures	DbInstanceName	Number of failed task executions	Coun	Alert: > 1% of task executions Critical: Increasing failure rate	

Understanding Key Metric Relationships

IOPS and Throughput Relationship

The 30% Headroom Rule: Always maintain at least **30% headroom** between your sustained operations per second and your provisioned IOPS. This provides buffer for:

- Compaction operations (can spike IOPS significantly)
- Any database restart to run smoothly
- Query bursts during peak usage
- Write spikes from batch ingestion
- Index maintenance operations

Example Calculation:

- Provisioned IOPS: 12,000
- Target Maximum Sustained IOPS (TotalIOpsPerSec): 8,400 (70% utilization)
- Reserved Headroom: 3,600 IOPS (30%)

If TotalIOpsPerSec consistently exceeds 8,400: → Upgrade storage tier or optimize workload

Monitoring Formula:

IOPS Utilization % = (ReadOpsPerSec + WriteOpsPerSec) / Provisioned IOPS × 100

- Target: Keep IOPS Utilization < 70%
- Warning: IOPS Utilization > 70%
- Critical: IOPS Utilization > 90%

Understanding Series Cardinality Performance Impact

Series cardinality has a multiplicative effect on system resources:

Series Count	Memory Impact	Query Performance Impact	Index Size Impact	Recommendation
< 100K	Minimal	Negligible	Small	Standard configuration
100K - 1M	Moderate	10-20% slower	Medium	Tune cache settings
1M - 5M	Significant	30-50% slower	Large	Aggressive optimization required
5M - 10M	High	50-70% slower	Very Large	Maximum tuning, consider redesign
> 10M	Severe	70%+ slower	Excessive	**Migrate to InfluxDB 3.0**

Why 10M is the Critical Threshold:

- InfluxDB 2.x architecture uses in-memory indexing
- Beyond 10M series, index operations become prohibitively expensive
- Memory requirements grow non-linearly
- Query planning overhead increases dramatically
- InfluxDB 3.0 uses a columnar storage engine designed for high cardinality

Instance Sizing and Performance Guidelines

The following table provides guidance on appropriate instance sizing based on your series cardinality and workload characteristics:

Max Series Count	Writes (lines/sec)	Reads (queries/sec)	Recommended Instance	Storage Type	Use Case
< 100K	~50,000	< 10	db.influx.large	Influx IO Included 3K	Small deployments, development, testing
< 1M	~150,000	< 25	db.influx.2xlarge	Influx IO Included 3K	Small to medium production workloads
~1M	~200,000	~25	db.influx.4xlarge	Influx IO Included 3K	Medium production workloads
< 5M	~250,000	~35	db.influx.4xlarge	Influx IO Included 12K	Large production workloads
< 10M	~500,000	~50	db.influx.8xlarge	Influx IO Included 12K	Very large production workloads
~10M	< 750,000	< 100	db.influx.12xlarge	Influx IO Included 12K	Maximum InfluxDB 2.x capacity
> 10M	N/A	N/A	**Migrate to InfluxDB 3.0**	N/A	Beyond InfluxDB 2.x optimal range

Configuration Optimization by Metric

High CPU Utilization (CPUUtilization > 70%)

Symptoms:

- **CPUUtilization** > 70% sustained
- **QueryRequestsTotal** (high volume or slow queries)
- **ActiveTaskWorkers** (high task load)

Configuration Adjustments:

Priority 1: Control Query Concurrency

- query-concurrency: Set to 50-75% of vCPU count
- Example: 8 vCPU instance → query-concurrency = 4-6

Priority 2: Limit Query Complexity

- influxql-max-select-series: 10000 (prevent unbounded queries)
- influxql-max-select-point: 100000000
- query-queue-size: 2048 (prevent queue buildup)

Priority 3: Enable Query Analysis

- flux-log-enabled: TRUE (temporarily for debugging)
- log-level: info (or debug for detailed analysis)

Important Considerations:

Reducing `query-concurrency` will limit the number of queries that can execute simultaneously, which may increase queued queries and lead to higher query latency during peak periods. Users may experience slower dashboard loads or report timeouts if query demand exceeds the reduced concurrency limit.

Setting protective limits (`influxql-max-select-series`, `influxql-max-select-point`) will cause queries that exceed these thresholds to fail with **compile_error** or **runtime_error** in

QueryRequestsTotal. While this protects the system from resource exhaustion, it may break existing queries that previously worked.

Best Practice: Before applying these changes, analyze your query patterns using **QueryResponseVolume** and **QueryRequestsTotal** metrics. Identify and optimize the most expensive queries first - look for queries without time range filters, queries spanning high-cardinality series, or queries requesting excessive data points. Optimizing queries at the application level is always preferable to imposing hard limits that may break functionality.

Hardware Actions:

- Scale to next instance class with more vCPUs
- Review query patterns for optimization opportunities

High Memory Utilization (MemoryUtilization > 70%)

Symptoms:

- **MemoryUtilization** > 70% sustained
- **HeapMemoryUsage** trending upward
- **ActiveMemoryAllocation** showing spikes
- **SeriesCardinality** (high cardinality increases memory usage)

Configuration Adjustments:

Priority 1: Reduce Cache Memory

- storage-cache-max-memory-size: Set to 10-15% of total RAM
- Example: 32GB RAM → 3,355,443,200 to 5,033,164,800 bytes
- storage-cache-snapshot-memory-size: 26,214,400 (25MB)

Priority 2: Limit Query Memory

- query-memory-bytes: Set to 60-70% of total RAM
- query-max-memory-bytes: Same as query-memory-bytes
- query-initial-memory-bytes: 10% of query-memory-bytes

Priority 3: Optimize Series Cache

- storage-series-id-set-cache-size: Reduce if high cardinality
- High memory: 100-200
- Normal: 500-1000

Important Considerations:

While these changes will reduce memory pressure, they will have a direct negative impact on application performance. Reducing `storage-cache-max-memory-size` means less data is cached in memory, forcing more disk reads and increasing query latency - you'll likely see **ReadOpsPerSec** increase and **QueryResponseVolume** response times degrade.

Limiting `query-memory-bytes` will cause memory-intensive queries to fail with **runtime_error** in **QueryRequestsTotal**, particularly queries that aggregate large datasets or return substantial result sets. Users may encounter "out of memory" errors for queries that previously succeeded.

Reducing `storage-series-id-set-cache-size` degrades performance for queries against high-cardinality data, as the system must recalculate series results more frequently instead of retrieving them from cache. This particularly impacts dashboards that repeatedly query the same series combinations.

Best Practice: Before applying these restrictive changes, analyze your query patterns and optimize them first:

- Review **QueryResponseVolume** to identify queries returning excessive data
- Use **QueryRequestsTotal** to find frequently executed queries that could benefit from optimization
- Add time range filters to reduce data scanning to what's necessary for your workload
- Implement query result caching at the application level
- Consider pre-aggregating data using downsampling tasks
- Review **SeriesCardinality** and optimize your data model to reduce unnecessary tags

Query optimization should always be your first approach - configuration restrictions should be a last resort when optimization isn't sufficient.

Hardware Actions:

- Increase instance size for more RAM

High Storage Utilization (DiskUtilization > 70%)

CloudWatch Metrics to Monitor:

- **DiskUtilization** > 70%
- **WriteThroughput** patterns
- **TotalBuckets** (many buckets increase overhead)

Configuration Adjustments:

Priority 1: Check Logging Configuration

- log-level: Ensure set to "info" (not "debug")
- flux-log-enabled: Set to FALSE unless actively debugging

Priority 2: Aggressive Retention

- storage-retention-check-interval: 15m0s (more frequent cleanup)

Priority 3: Optimize Compaction

- storage-compact-full-write-cold-duration: 2h0m0s (more frequent)
- storage-cache-snapshot-write-cold-duration: 5m0s

Priority 4: Reduce Index Size

- storage-max-index-log-file-size: 524,288 (512KB for faster compaction)

Important Considerations:

Critical First Step - Check Your Logging Configuration: Before making any other changes, verify your logging settings. **Debug logging and Flux query logs can consume as much or more disk space than your actual time-series data**, and this is one of the most common causes of unexpected storage exhaustion.

Logging Impact:

- `log-level: debug` generates extremely verbose logs, potentially hundreds of MB per hour
- `flux-log-enabled: TRUE` logs every Flux query execution with full details, creating massive log files
- These logs accumulate rapidly and are often overlooked during capacity planning
- Log files can fill disk space faster than data ingestion, especially on smaller instances
- Unlike time-series data, logs are kept in local storage for 24 hours before deletion

Immediate Actions if Logs are Large:

1. Set `log-level: info` (from debug)
2. Set `flux-log-enabled: FALSE`
3. Monitor **DiskUtilization** for immediate improvement

Compaction Configuration Trade-offs:

These configuration changes are specifically designed for workloads with **high ingestion throughput and short retention windows** where disk usage fluctuates substantially. They force the compaction engine to work more aggressively, which is only beneficial in specific scenarios.

Critical Trade-offs: Increasing compaction frequency will significantly increase resource consumption:

- **CPUUtilization** will rise as compaction operations consume CPU cycles
- **MemoryUtilization** will increase during compaction as data is loaded and processed
- **WriteOpsPerSec** and **WriteThroughput** will spike during compaction windows, potentially exceeding your 30% IOPS headroom
- **WriteTimeouts** may increase if compaction I/O competes with application writes

These changes can create a cascading performance problem where aggressive compaction consumes resources needed for query and write operations, degrading overall system performance even while reducing disk usage.

Best Practice: Before adjusting compaction settings, focus on data and logging management:

1. **Check Logging First (Most Common Issue):** Verify log-level is "info" and flux-log-enabled is FALSE
2. **Review Your Data Model:** Are you writing data you don't actually need? Can you reduce measurement or field granularity?
3. **Optimize Retention Policies:** Check **TotalBuckets** and review retention settings for each bucket
4. **Monitor Compaction Impact:** Baseline your **CPUUtilization**, **MemoryUtilization**, and **WriteOpsPerSec** before changes

Alternative Approaches:

- Increase storage capacity (often simpler and more cost-effective)
- Implement data downsampling or aggregation strategies
- Consolidate buckets (reduce **TotalBuckets**) to decrease overhead
- Review and enforce retention policies more strictly

Only apply aggressive compaction settings if you've optimized data management and confirmed your instance has sufficient CPU, memory, and IOPS headroom to handle the increased load.

Hardware Actions:

- Increase storage capacity

High IOPS Utilization (ReadIOPS/WriteIOPS/TotalOperationsPerSecond > 70% of provisioned)

CloudWatch Metrics to Monitor:

- **ReadOpsPerSec** + **WriteOpsPerSec** = **TotalIOpsPerSec**
- **ReadThroughput** and **WriteThroughput**
- Compare against provisioned IOPS (3K, 12K, or 16K)

Configuration Adjustments:

Priority 1: Control Compaction I/O

- storage-max-concurrent-compactions: 2-3 (limit concurrent compactions)

- storage-compact-throughput-burst: Adjust based on disk capability
- 3K IOPS: 25,165,824 (24MB/s)
- 12K IOPS: 50,331,648 (48MB/s)

Priority 2: Optimize Write Operations

- storage-wal-max-concurrent-writes: 8-12
- storage-wal-max-write-delay: 5m0s

Priority 3: Adjust Snapshot Timing

- storage-cache-snapshot-write-cold-duration: 15m0s (less frequent)
- storage-compact-full-write-cold-duration: 6h0m0s (less frequent)

Important Considerations:

These changes create significant trade-offs between I/O utilization and system performance:

Limiting Compaction I/O:

- Reducing `storage-max-concurrent-compactions` will slow down compaction operations, causing TSM files to accumulate and **DiskUtilization** to increase more rapidly
- Lower `storage-compact-throughput-burst` extends compaction duration, keeping the compactor active longer and potentially blocking other operations
- Slower compaction means query performance degrades over time as the storage engine must read from more, smaller TSM files instead of consolidated ones
- You may see **QueryRequestsTotal** runtime_error rates increase as queries timeout while waiting for I/O

Reducing Snapshot Frequency:

- Increasing `storage-cache-snapshot-write-cold-duration` and `storage-compact-full-write-cold-duration` means data stays in the write-ahead log (WAL) longer
- This increases **MemoryUtilization** as more data is held in cache before being flushed to disk
- Risk of data loss increases slightly if the instance crashes before cached data is persisted

- Recovery time after a restart increases as more WAL data must be replayed

Write Operation Tuning:

- Reducing `storage-wal-max-concurrent-writes` will serialize write operations more, potentially increasing **WriteTimeouts** during high-throughput periods
- Increasing `storage-wal-max-write-delay` means writes may wait longer before being rejected, which can mask capacity problems but frustrate users with slow responses

Best Practice: High IOPS utilization usually indicates you've outgrown your storage tier rather than a configuration problem. Before restricting I/O, analyze I/O patterns and optimize before restricting.

Hardware Actions:

- Upgrade to higher IOPS storage tier (3K → 12K)
- Ensure 30% IOPS headroom is maintained

High Series Cardinality (SeriesCardinality > 1M)

CloudWatch Metrics to Monitor:

- **SeriesCardinality** per bucket and total
- **MemoryUtilization** (increases with cardinality)
- **CPUUtilization** (query planning overhead)
- **QueryRequestsTotal** (runtime_error rate may increase)

Configuration Adjustments:

Priority 1: Optimize Series Handling

- storage-series-id-set-cache-size: 1000-2000 (increase cache)
- storage-series-file-max-concurrent-snapshot-compactions: 4-8

Priority 2: Set Protective Limits

- influxql-max-select-series: 10000 (prevent runaway queries)

- influxql-max-select-buckets: 1000

Priority 3: Optimize Index Operations

- storage-max-index-log-file-size: 2,097,152 (2MB)

Important Considerations:

High series cardinality is fundamentally a data modeling problem, not a configuration problem. Configuration changes can only mitigate symptoms - they cannot solve the underlying issue.

Configuration Trade-offs:

Increasing `storage-series-id-set-cache-size` will improve query performance by caching series lookups, but at the cost of increased **MemoryUtilization**. Each cache entry consumes memory, and with millions of series, this can be substantial. Monitor **HeapMemoryUsage** and **ActiveMemoryAllocation** after making this change.

Setting protective limits (`influxql-max-select-series`, `influxql-max-select-buckets`) will cause legitimate queries to fail with **compile_error** in **QueryRequestsTotal** if they exceed these thresholds. Dashboards that previously worked may break, and users will need to modify their queries. This is particularly problematic for:

- Monitoring dashboards that aggregate across many hosts/services
- Analytics queries that need to compare multiple entities
- Alerting queries that evaluate fleet-wide conditions

Adjusting `storage-max-index-log-file-size` to smaller values increases index compaction frequency, which raises **CPUUtilization** and **WriteOpsPerSec** as the system performs more frequent index maintenance.

Critical Understanding:

When **SeriesCardinality** exceeds 5M, you're approaching the architectural limits of InfluxDB 2.x. At 10M+ series, performance degrades exponentially regardless of configuration:

- Query planning becomes prohibitively expensive (high **CPUUtilization**)
- Memory requirements grow non-linearly (high **MemoryUtilization**)

- Index operations dominate I/O (**ReadOpsPerSec**, **WriteOpsPerSec**)
- **QueryRequestsTotal** runtime_error rates increase as queries timeout or exhaust memory

Best Practice: Configuration changes are temporary band-aids. You must address the root cause:

1. **Analyze Your Data Model:**
   - Review **SeriesCardinality** per bucket to identify problem areas
   - Identify which tags have high unique value counts
   - Look for unbounded tag values (UUIDs, timestamps, user IDs, session IDs)
   - Find tags that should be fields instead

Data Model Actions:

- Review tag design to reduce unnecessary cardinality
- Consider consolidating similar series
- **If > 10M series:** Plan migration to InfluxDB 3.0

Query Performance Issues

CloudWatch Metrics to Monitor:

- **QueryRequestsTotal** by result type (success, runtime_error, compile_error, queue_error)
- **APIRequestRate** with Status=500 or Status=499
- **QueryResponseVolume** (large responses indicate expensive queries)

Configuration Adjustments:

Priority 1: Increase Query Resources

- query-concurrency: Increase to 75% of vCPUs
- query-memory-bytes: Allocate 70% of total RAM
- query-queue-size: 4096

Priority 2: Optimize Query Execution

- storage-series-id-set-cache-size: 1000 (increase for better caching)
- http-read-timeout: 60s (prevent premature timeouts)

Priority 3: Set Reasonable Limits

- influxql-max-select-point: 100000000
- influxql-max-select-series: 10000
- influxql-max-select-buckets: 1000

Important Considerations:

Increasing query resources creates resource competition and potential system instability:

Resource Allocation Trade-offs:

Increasing `query-concurrency` allows more queries to run simultaneously, but each query competes for CPU and memory:

- **CPUUtilization** will increase, potentially reaching saturation during peak query periods
- **MemoryUtilization** will rise as more queries allocate memory simultaneously
- If you increase concurrency without adequate resources, all queries slow down instead of just some queuing
- Risk of cascading failure if concurrent queries exhaust available resources

Allocating more `query-memory-bytes` means less memory available for caching and other operations:

- **HeapMemoryUsage** will increase
- `storage-cache-max-memory-size` may need to be reduced to compensate
- Fewer cache hits means higher **ReadOpsPerSec** and slower query performance
- System becomes more vulnerable to memory exhaustion if queries use their full allocation

Increasing `query-queue-size` only delays the problem - it doesn't solve capacity issues:

- Queries wait longer in queue, increasing end-to-end latency
- Users perceive the system as slower even though throughput may be unchanged

- Large queues can mask underlying capacity problems
- **QueryRequestsTotal** queue_error rate decreases, but user experience may not improve

Increasing `http-read-timeout` prevents premature query cancellation, but:

- Long-running queries consume resources longer, reducing capacity for other queries
- Users wait longer before receiving timeout errors
- Can hide inefficient queries that should be optimized
- May lead to resource exhaustion if many slow queries accumulate

Best Practice: Query performance problems are usually caused by inefficient queries, not insufficient resources. Before increasing resource allocation:

1. **Analyze Query Patterns:**
   - Review **QueryResponseVolume** to identify queries returning excessive data (> 1MB)
   - Check **QueryRequestsTotal** runtime_error patterns - what's causing failures?
   - Look for **APIRequestRate** with Status=499 (client timeouts) - queries are too slow
   - Identify frequently executed expensive queries
2. **Optimize Queries First:**

   Common Query Anti-patterns:
   - Missing time range filters → Add explicit time bounds
   - Querying all series → Add specific tag filters
   - Excessive aggregation windows → Use appropriate intervals
   - Unnecessary fields in SELECT → Request only needed data
   - No LIMIT clauses → Add reasonable limits
3. **Application-Level Solutions:**
   - Implement query result caching (Redis, Memcached)
   - Use tasks to pre-aggregate common patterns
   - Add pagination for large result sets
   - Implement query rate limiting per user/dashboard
   - Use downsampled data for historical queries
4. **Verify Resource Availability:**

- Check **CPUUtilization** - if already > 70%, increasing concurrency will make things worse
- Check **MemoryUtilization** - if already > 70%, allocating more query memory will cause OOM
- Verify **TotalIOpsPerSec** has 30% headroom before increasing query load

Recommended Approach:

1. Start by optimizing the top 10 most expensive queries (by **QueryResponseVolume**)
2. Implement query result caching at the application level
3. Only increase resource allocation if queries are optimized and metrics show headroom
4. Scale to a larger instance class if workload has outgrown current capacity

Hardware Actions:

- Scale your compute capacity, queries benefit from extra processing power (vCPUs)

RegEx Performance Pitfalls in Flux Queries

When filtering data in Flux, avoid using regular expressions for exact matches or simple pattern matching, as this introduces significant performance penalties. RegEx operations in Flux are **single-threaded** and **bypass the underlying TSM index entirely**. Instead of leveraging InfluxDB's optimized tag indexes for fast lookups, RegEx filters force the query engine to retrieve all matching series from storage and perform text comparisons sequentially against each value. This becomes particularly problematic when:

- **Filtering on exact tag values** - Use the equality operator (==) or the `contains()` function instead of RegEx patterns like `/^exact_value$/`
- **Matching multiple specific values** - Use the `in` operator with an array of values rather than alternation patterns like `/(value1|value2|value3)/`
- **Simple prefix or suffix matching** - Consider using `strings.hasPrefix()` or `strings.hasSuffix()` functions, which are more efficient than RegEx anchors

For scenarios requiring multiple pattern matches, restructure your query to use multiple filter predicates combined with logical operators, or pre-filter using tag equality before applying more complex string operations. Reserve RegEx exclusively for cases requiring true pattern matching that cannot be expressed through simpler comparison operators.

Write Performance Issues

CloudWatch Metrics to Monitor:

- **WriteTimeouts** (increasing count)
- **WriteOpsPerSec** and **WriteThroughput**
- **APIRequestRate** with Status=500 for write endpoints
- **QueryRequestsTotal** with result=runtime_error during writes

Configuration Adjustments:

Priority 1: Optimize WAL Writes

- storage-wal-max-concurrent-writes: 12-16
- storage-wal-max-write-delay: 10m0s
- http-write-timeout: 60s

Priority 2: Optimize Cache Snapshots

- storage-cache-snapshot-memory-size: 52,428,800 (50MB)
- storage-cache-snapshot-write-cold-duration: 10m0s

Priority 3: Control Field Validation

- storage-no-validate-field-size: TRUE (if data source is trusted)

Important Considerations:

Write performance tuning involves careful trade-offs between throughput, reliability, and resource consumption:

WAL Configuration Trade-offs:

Increasing `storage-wal-max-concurrent-writes` allows more parallel write operations, but:

- **CPUUtilization** increases as more write threads compete for CPU
- **MemoryUtilization** rises as more data is buffered in memory before WAL flush

- **WriteOpsPerSec** will spike, potentially exceeding your 30% IOPS headroom
- Increased contention for disk I/O may actually slow down individual writes
- If you exceed disk I/O capacity, **WriteTimeouts** may increase rather than decrease

Increasing `storage-wal-max-write-delay` means writes wait longer before timing out:

- Masks capacity problems by making writes wait instead of failing quickly
- Users experience slower write response times even when writes eventually succeed
- Can lead to write queue buildup and memory pressure
- Doesn't actually increase capacity - just delays the timeout

Increasing `http-write-timeout` similarly delays timeout errors:

- Allows larger batch writes to complete
- But also allows slow writes to consume resources longer
- Can hide underlying performance problems
- May lead to resource exhaustion if many slow writes accumulate

Cache Snapshot Trade-offs:

Increasing `storage-cache-snapshot-memory-size` means more data accumulates in memory before flushing:

- **MemoryUtilization** increases significantly
- Risk of data loss increases if instance crashes before snapshot
- Larger snapshots take longer to write, creating bigger **WriteOpsPerSec** spikes
- Can improve write throughput by batching more data, but at cost of memory and reliability

Increasing `storage-cache-snapshot-write-cold-duration` delays snapshots:

- Further increases **MemoryUtilization** as data stays in cache longer
- Increases data loss risk window
- Reduces **WriteOpsPerSec** frequency but creates larger spikes when snapshots occur

- Recovery time after restart increases as more WAL must be replayed

Field Validation Trade-off:

Setting `storage-no-validate-field-size: TRUE` disables field size validation:

- Improves write throughput by skipping validation checks
- **Critical Risk:** Allows malformed or malicious data to be written
- Can lead to data corruption if writes contain invalid field sizes
- Makes debugging data problems much harder
- **Only use if you have complete control and trust of your data source**

Best Practice: Write performance problems usually indicate capacity limits or inefficient write patterns. Before tuning configuration:

1. **Analyze Write Patterns:**
   - Review **WriteThroughput** and **WriteOpsPerSec** trends
   - Check **WriteTimeouts** correlation with write load
   - Monitor **APIRequestRate** for write endpoints by status code
   - Identify write batch sizes and frequency
2. **Optimize Write Operations First:**

   Common Write Anti-patterns:

   - Writing individual points → Batch writes (5,000-10,000 points)
   - Too-frequent writes → Buffer and batch
   - Synchronous writes → Implement async write queues
   - Unbounded write bursts → Implement rate limiting
   - Writing unnecessary precision → Round timestamps appropriately
3. **Verify I/O Capacity:**
   - Check **TotalIOpsPerSec** - if already > 70%, increasing WAL concurrency will make things worse
   - Review **WriteOpsPerSec** during peak periods
   - Ensure 30% IOPS headroom exists before tuning write settings
   - Consider whether 3K IOPS is sufficient or if 12K IOPS tier is needed

4. **Application-Level Improvements:**
   - Implement write buffering with configurable batch sizes
   - Add write retry logic with exponential backoff
   - Use asynchronous write operations
   - Implement write rate limiting during peak periods
   - Monitor write queue depth and apply backpressure

Recommended Approach:

1. Start by optimizing write batch sizes at the application level (aim for 5,000-10,000 points per batch)
2. Implement write buffering and async operations
3. Verify **TotalIOpsPerSec** has adequate headroom
4. Upgrade to the next storage tier (3K IOPS → 12K IOPS → 16K IOPS) if consistently above 70% utilization
5. Only tune WAL settings if writes are optimized and I/O capacity is adequate
6. **Never** disable field validation unless you have complete control of data sources

Hardware Actions:

- Upgrade to higher IOPS storage (3K → 12K → 16K)
- Ensure I/O headroom is adequate
- Scale to larger instance class if CPU or memory constrained

Monitoring Best Practices

CloudWatch Alarms Configuration

Critical Alarms (Immediate Action Required):

CPUUtilization:

- Threshold: > 90% for 5 minutes
- Action: Implement traffic remediation measures or Compute Scaling

MemoryUtilization:

- Threshold: > 90% for 5 minutes
- Action: Implement traffic remediation measures or Compute Scaling

DiskUtilization:

- Threshold: > 85%
- Action: Try to free up space by deleting old buckets, updating retention configurations or Storage Scaling

TotalIOpsPerSec:

- Threshold: > 90% of provisioned for 10 minutes
- Action: Implement traffic remediation measures or Increase IOPS

SeriesCardinality:

- Threshold: > 10,000,000
- Action: Review your Data model, if no changes are possible explore migrate to InfluxDB 3 or shard your data

EngineUptime:

- Threshold: Unexpected reset (< 300 seconds)
- Action: Check is it coincides with a maintenance window, if not create a ticket to Timestream support.

Warning Alarms (Investigation Required):

CPUUtilization:

- Threshold: > 70% for 15 minutes
- Action: review changes in workload or traffic

MemoryUtilization:

- Threshold: > 70% for 15 minutes
- Action: review changes in workload or traffic

DiskUtilization:

- Threshold: > 70%
- Action: Review retention policies

TotalIOpsPerSec:

- Threshold: > 70% of provisioned for 15 minutes
- Action: review changes in workload or traffic

QueryRequestsTotal (runtime_error):

- Threshold: > 1% of total queries
- Action: review changes in workload or traffic

WriteTimeouts:

- Threshold: > 1% of write operations
- Action: review changes in workload or traffic

SeriesCardinality:

- Threshold: > 5,000,000
- Action: Review data model optimization

Proactive Monitoring Checklist

Daily:

- Review APIRequestRate for error spikes (400, 404, 499, 500)
- Check QueryRequestsTotal for runtime_error and queue_error rates
- Verify WriteTimeouts count is minimal

- Check for any critical alarms
- Verify EngineUptime (no unexpected restarts)

Weekly:

- Review CPUUtilization, MemoryUtilization, and DiskUtilization trends
- Analyze QueryRequestsTotal patterns by result type
- Check SeriesCardinality growth rate per bucket
- Review TotalIOpsPerSec utilization trends
- Verify configuration parameters are optimal
- Review TaskExecutionFailures patterns

Monthly:

- Capacity planning review (project 3-6 months ahead)
- Compare current metrics against sizing table
- Review and optimize retention policies
- Analyze query patterns from APIRequestRate and QueryResponseVolume
- Review SeriesCardinality and data model efficiency
- Assess need for instance scaling or configuration changes
- Review TotalBuckets and consolidation opportunities

Troubleshooting Guide

Scenario: Sudden Performance Degradation

Investigation Steps:

Check Recent Changes:

- Configuration parameter modifications in the AWS Management Console
- Application deployment changes
- Query pattern changes

- Data model modifications
- Infrastructure changes (instance type, storage)

Review CloudWatch Metrics:

- **CPU spike?** → Check CPUUtilization, QueryRequestsTotal
- **Memory pressure?** → Check MemoryUtilization, HeapMemoryUsage, ActiveMemoryAllocation
- **IOPS saturation?** → Check TotalIOpsPerSec, ReadOpsPerSec, WriteOpsPerSec
- **Series cardinality jump?** → Check SeriesCardinality growth
- **Error rate increase?** → Check QueryRequestsTotal (runtime_error), APIRequestRate (Status=500)
- **Unexpected restart?** → Check EngineUptime

Enable Detailed Logging:

Configuration changes:

- log-level: debug
- flux-log-enabled: TRUE

Monitor for 1-2 hours, then review logs

Return to log-level: info after investigation

Resolution Steps:

- Apply appropriate configuration changes based on findings
- Scale resources if limits are reached
- Optimize queries or data model if needed
- Implement rate limiting if sudden load increase

Scenario: Memory Exhaustion

Symptoms:

- MemoryUtilization > 90%

- HeapMemoryUsage approaching maximum
- QueryRequestsTotal showing runtime_error (out of memory)
- APIRequestRate showing Status=500

Resolution Steps:

Immediate Actions (if critical):

1. Restart instance to clear memory (if safe to do so)
2. Reduce query-concurrency temporarily
3. Eliminate long-running queries if possible

Configuration Changes:

Priority 1: Reduce Cache Memory

- storage-cache-max-memory-size: Reduce to 10% of RAM
- Example: 32GB → 3,355,443,200 bytes
- storage-cache-snapshot-memory-size: 26,214,400 (25MB)

Priority 2: Limit Query Memory

- query-memory-bytes: Set to 60% of total RAM
- query-max-memory-bytes: Match query-memory-bytes
- query-initial-memory-bytes: 10% of query-memory-bytes

Priority 3: Set Protective Limits

- influxql-max-select-series: 10000
- influxql-max-select-point: 100000000
- query-concurrency: Reduce to 50% of vCPUs

Long-Term Solutions:

- Optimize data model to reduce **SeriesCardinality**

- Implement query result size limits at application level
- Add query timeout enforcement
- Review most common queries to ensure these are following best practices mentioned in the section the section called "Query Performance Issues"

Scenario: High Series Cardinality Impact

Review CloudWatch metrics:

- **SeriesCardinality** > 5M
- **MemoryUtilization** high
- **QueryRequestsTotal** showing increased runtime_error
- **CPUUtilization** elevated due to query planning overhead

Investigation Steps:

Analyze Cardinality Growth:

- SeriesCardinality growth rate (daily/weekly)
- Projection to 10M threshold
- Identify sources of high cardinality
- Review tag design and usage

Assess Performance Impact:

- Compare **QueryRequestsTotal** success rate before/after cardinality increase
- Review **MemoryUtilization** correlation
- Check **CPUUtilization** patterns
- Analyze **QueryResponseVolume** trends

Identify Cardinality Sources:

Review data model:

- Which buckets have highest SeriesCardinality?

- Which tags have high unique value counts?
- Are there unnecessary tags?
- Are tag values unbounded (UUIDs, timestamps, etc.)?

Review Current Configuration:

Check optimization parameters:

- storage-series-id-set-cache-size: Current value?
- influxql-max-select-series: Is it limiting runaway queries?
- storage-max-index-log-file-size: Appropriate for cardinality?

Resolution Steps:

Immediate Configuration Changes:

Priority 1: Optimize Series Handling

- storage-series-id-set-cache-size: 1500-2000
- storage-series-file-max-concurrent-snapshot-compactions: 6-8
- storage-max-index-log-file-size: 2,097,152 (2MB)

Priority 2: Set Protective Limits

- influxql-max-select-series: 10000
- influxql-max-select-buckets: 1000
- query-concurrency: Reduce if memory constrained

Priority 3: Increase Resources

- Scale to next instance tier
- Increase memory allocation
- Consider 12K IOPS storage tier

Migration Planning (if > 10M series):

- **InfluxDB 3.0 offers superior high-cardinality performance**
- Plan migration timeline (2-3 months)
- Test with subset of data first
- Prepare application for migration
- InfluxDB 3.0 uses columnar storage optimized for billions of series

Scenario: Query Queue Buildup

Review CloudWatch metrics:

- **QueryRequestsTotal** with result=queue_error increasing (queries being rejected)
- **APIRequestRate** with Status=429 or Status=503 (service unavailable/too many requests)
- **CPUUtilization** may be elevated (> 70%) indicating resource saturation
- **MemoryUtilization** may be high (> 70%) limiting query capacity
- **QueryResponseVolume** showing large response sizes (queries taking excessive resources)

Investigation Steps:

Analyze Queue and Concurrency Metrics:

- Review **QueryRequestsTotal** breakdown by result type:
  - High queue_error count indicates queries are being rejected
  - Compare success rate to baseline - is it dropping?
  - Check for runtime_error increases (queries failing after starting)
- Monitor **APIRequestRate** patterns:
  - Look for Status=429 (too many requests) or Status=503 (service unavailable)
  - Identify which endpoints are experiencing rejections
  - Check request rate trends over time

Review Resource Utilization:

- **CPUUtilization** during high queue periods:
  - If > 70%, queries are CPU-bound and can't execute faster
  - If < 50%, queue limits may be too restrictive

- **MemoryUtilization** correlation:
  - High memory may be limiting query concurrency
  - Check **HeapMemoryUsage** and **ActiveMemoryAllocation** for memory pressure
- **TotalIOpsPerSec** patterns:
  - High I/O may be slowing query execution
  - Check if queries are I/O bound

Identify Query Patterns:

- Review **QueryResponseVolume**:
  - Are queries returning excessive data (> 1MB)?
  - Identify endpoints with largest response volumes
  - Look for patterns in expensive queries
- Analyze **QueryRequestsTotal** rate:
  - What's the queries per second rate?
  - Are there burst patterns or sustained high load?
  - Compare to instance capacity from sizing table
- Check **APIRequestRate** by endpoint:
  - Which query endpoints have highest traffic?
  - Are there duplicate or redundant queries?

Check Resource Availability:

- Compare current metrics to sizing table recommendations:
  - **SeriesCardinality** vs. instance class capacity
  - Query rate vs. recommended queries per second
  - **CPUUtilization** and **MemoryUtilization** headroom
- Verify IOPS capacity:
  - **TotalIOpsPerSec** should have 30% headroom
  - Check if queries are waiting on disk I/O

Resolution Steps:

Configuration Changes:

Priority 1: Increase Queue Capacity

- query-queue-size: 4096 (from default 1024)

Priority 2: Increase Concurrency (if resources allow)

- query-concurrency: Increase to 75% of vCPUs
- Example: 16 vCPU → query-concurrency = 12
- Verify CPUUtilization stays < 80% after change
- Verify MemoryUtilization stays < 80% after change

Priority 3: Optimize Query Execution

- query-memory-bytes: Ensure adequate allocation
- storage-series-id-set-cache-size: 1000-1500
- http-read-timeout: 120s (prevent premature timeouts)

Priority 4: Set Protective Limits

- influxql-max-select-series: 10000
- influxql-max-select-point: 100000000

Application-Level Solutions:

- **Implement query result caching** (Redis, Memcached)
  - Cache results for frequently executed queries
  - Set appropriate TTLs based on data freshness requirements
  - Monitor cache hit rates
- **Use continuous queries** to pre-aggregate common patterns
  - Pre-calculate common aggregations
  - Query pre-aggregated data instead of raw data
- **Add pagination** for large result sets

  - Limit initial query size
  - Load additional data on demand
- **Implement query rate limiting** per user/dashboard
  - Prevent single users from overwhelming the system
  - Set fair-use quotas
- **Use downsampled data** for historical queries
  - Query lower-resolution data for older time ranges
  - Reserve full-resolution queries for recent data

Scaling Decision:

- If CPUUtilization > 70% sustained: Scale to larger instance
- If MemoryUtilization > 70% sustained: Scale to memory-optimized instance
- If query rate exceeds instance capacity: Scale to next tier per sizing table

Customer-managed backup and restore

Customer-managed backup and restore lets you create, manage, and restore database backups for your Amazon Timestream for InfluxDB DB instances and clusters. You can create on-demand or automated scheduled backups, and restore databases without opening AWS support tickets.

Overview

Customer-managed backup and restore provides self-service backup capabilities for Amazon Timestream for InfluxDB:

- **InfluxDB v2** (standalone instances)
- **InfluxDB v2 Read Replicas** (multi-node clusters)

For InfluxDB v3 backup and restore, see the section called “Backup and restore”.

The first backup creates a complete copy of your database. Subsequent backups are incremental and capture only the data that changed since the last backup, reducing storage costs and backup time.

Key concepts

Concept	Description
DbBackup	A backup of your database containing the instance or cluster configuration at the time of backup.
DbBackupConfiguration	Automated recurring backups at a defined frequency and retention.
Restore	Creates a new DB from a DbBackup or restores an existing resource back to the provided DbBackup.
RetentionDays	Defines the number of days a backup is kept before automatic deletion.
On Demand Backup	Triggers a one-time backup of the database at a point in time.
Custom Schedule Backup	Defines a custom AWS cron schedule when the DB will be backed up automatically.

Creating a DbBackup

On-demand backup (AWS CLI)

Create an on-demand backup of a DB instance or cluster:

```
aws timestream-influxdb create-db-backup \
    --db-resource-id my-resource-id \
    --name my-db-backup-2026-07-07 \
    --retention-days 7
```

For Read Replicas clusters, use the DB cluster ID as the resource identifier.

On-demand backup (AWS Management Console)

1. Open the Amazon Timestream for InfluxDB console.
2. Choose your database instance or cluster.

3. Choose **Actions**, **Create backup**.
4. Enter a backup name and retention period.
5. Choose **Create backup**.

Configuring automated backups

Automated backups can be enabled at resource creation time or updated afterwards. Only one DbBackupConfiguration is allowed per type per resource, and a maximum of 4 DbBackupConfigurations can be enabled per resource.

Enable automated backups on create (AWS CLI)

For a DB instance:

```
aws timestream-influxdb create-db-instance \
    --name my-db-instance \
    --db-instance-type db.influx.medium \
    --db-storage-type InfluxIOIncludedT1 \
    --allocated-storage 100 \
    --vpc-subnet-ids subnet-0123456789abcdef0 \
    --vpc-security-group-ids sg-0123456789abcdef0 \
    --password MySecurePassword123! \
    --db-backup-configurations '[{
       "enabled": true,
       "type": "HOURLY",
       "retentionDays": 1
      },
      {
       "enabled": true,
       "type": "DAILY",
       "retentionDays": 7
      },
      {
       "enabled": true,
       "type": "WEEKLY",
       "retentionDays": 30
      },
      {
       "enabled": true,
       "type": "MONTHLY",
       "retentionDays": 365
```

```
      }]'
```

For a Read Replicas cluster:

```
aws timestream-influxdb create-db-cluster \
    --name my-db-cluster \
    --db-instance-type db.influx.medium \
    --db-storage-type InfluxIOIncludedT1 \
    --allocated-storage 100 \
    --vpc-subnet-ids subnet-0123456789abcdef0 \
    --vpc-security-group-ids sg-0123456789abcdef0 \
    --password MySecurePassword123! \
    --deployment-type MULTI_NODE_READ_REPLICAS \
    --db-backup-configurations '[{
       "enabled": true,
       "type": "HOURLY",
       "retentionDays": 1
      },
      {
       "enabled": true,
       "type": "DAILY",
       "retentionDays": 7
      },
      {
       "enabled": true,
       "type": "WEEKLY",
       "retentionDays": 30
      },
      {
       "enabled": true,
       "type": "MONTHLY",
       "retentionDays": 365
      }]'
```

Enable automated backups on update (AWS CLI)

DbBackupConfigurations can be enabled, disabled, and retention can be changed after creation:

```
aws timestream-influxdb update-db-instance \
    --identifier my-db-instance-id \
    --db-backup-configurations '[{
       "enabled": true,
       "type": "HOURLY",
```

```
        "retentionDays": 1
      },
      {
       "enabled": true,
       "type": "DAILY",
       "retentionDays": 7
      },
      {
       "enabled": false,
       "type": "WEEKLY",
       "retentionDays": 30
      }]'
```

For a Read Replicas cluster:

```
aws timestream-influxdb update-db-cluster \
    --db-cluster-id my-db-cluster-id \
    --db-backup-configurations '[{
       "enabled": true,
       "type": "HOURLY",
       "retentionDays": 1
      },
      {
       "enabled": true,
       "type": "DAILY",
       "retentionDays": 7
      },
      {
       "enabled": false,
       "type": "WEEKLY",
       "retentionDays": 30
      }]'
```

DbBackupConfiguration types

Type	Description
HOURLY	Creates a DbBackup every hour
DAILY	Creates a DbBackup every day at midnight in the timezone of the Region

Type	Description
WEEKLY	Creates a DbBackup every week on Sunday at midnight in the timezone of the Region
MONTHLY	Creates a DbBackup every month on the 1st at midnight in the timezone of the Region
CUSTOM	Use an AWS Cron Expression for a custom frequency. Must be at a daily frequency or less.

Example CUSTOM type:

```
--db-backup-configurations '[{
     "enabled": true,
     "type": "CUSTOM",
     "retentionDays": 7,
     "customSchedule": "cron(00 18 * * * *)"
    }]'
```

This creates a backup that runs at 18:00 every day in the local timezone.

> **Note**
>
> Automated backups can start anytime within an hour of the scheduled time.

Best practices for scheduling

- Use shorter frequencies like HOURLY with shorter retention days.
- Use longer frequencies like WEEKLY or MONTHLY with longer retention days.
- If using a custom schedule, schedule the backup during low-traffic windows.

Listing and viewing backups

List all backups for a resource:

```
aws timestream-influxdb list-db-backups \
```

```
    --db-resource-id my-resource-id
```

List all backups in the account:

```
aws timestream-influxdb list-db-backups
```

Get backup details:

```
aws timestream-influxdb get-db-backup \
    --identifier my-db-backup-id
```

DbBackup statuses

Status	Description
IN_PROGRESS	Backup is being created (database remains available)
COMPLETED	Backup is completed and available to use for restore
FAILED	Backup creation failed
DELETING	Backup is being deleted
DELETED	Backup has been deleted

Restoring from a DbBackup

A DbBackup can be restored to a new resource, or it can replace the existing resource where the backup was taken.

Restore to a new resource

This is the default and recommended restore mode since it does not impact any existing resources.

```
aws timestream-influxdb restore-from-db-backup \
    --name my-new-restored-database \
    --db-backup-id my-backup-id \
    --restore-mode NEW_RESOURCE
```

If no additional parameters are provided, the new resource inherits all parameters from the DbBackup. A subset of these parameters can be overridden during restore if provided.

A backup of a DB instance can be restored to a Read Replicas cluster. A backup of a cluster cannot be restored to a standalone DB instance.

Restoring a standalone instance backup to a Read Replicas cluster

You can restore a backup from a standalone instance (Single-AZ or Multi-AZ) to a Read Replicas cluster. This operation is a **migration**, not a point-in-time restore, because of differences in the internal storage format.

> **Warning**
>
> Standalone instances use a Write-Ahead Log (WAL) that periodically flushes to TSM storage files. When restoring to a Read Replicas cluster, only data that has been persisted to TSM files is included. Data still in the WAL at the time of the backup is not restored. WAL data is flushed to TSM after approximately 10 minutes of write inactivity or when the in-memory cache reaches 25 MB. These thresholds are configurable via parameter groups.
> If your instance has never generated TSM files, the restore to a Read Replicas cluster results in an empty database with no data. This can occur when:
>
> - The instance is new and has received little or no data.
> - Write volume is very low (below 25 MB per shard) and writes arrive continuously without a 10-minute pause—preventing both the size-based and time-based flush triggers from firing.

To minimize data not yet persisted to TSM, stop writes to the source instance and wait at least 10 minutes before creating the backup. Restoring to another standalone instance (Single-AZ or Multi-AZ) does not have this limitation—all data including WAL contents is preserved.

Restore to an existing resource

```
aws timestream-influxdb restore-from-db-backup \
    --name my-existing-resource-name \
    --db-backup-id my-backup-id \
    --restore-mode REPLACE_EXISTING
```

When restoring to an existing resource, no additional parameters are accepted and no changes to the existing database configuration will be made besides restoring the data.

> **Warning**
>
> Restoring to an existing resource is a **destructive operation**. The resource will not be available during this operation. Existing data will be deleted and replaced with the data from the time of the backup. If there are failures during the operation, the service will attempt to roll back the resource with the original data. It is highly recommended to take an on-demand DbBackup prior to performing this action. After the operation completes, the original data is deleted and not recoverable unless a DbBackup was taken.

You can restrict this operation using an IAM policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": "timestream-influxdb:RestoreFromDbBackup",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "timestream-influxdb:RestoreMode": "REPLACE_EXISTING"
                }
            }
        }
    ]
}
```

Restore (AWS Management Console)

1. Open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **Backups**.
3. Select the backup you want to restore from.
4. Choose **Actions**, **Restore backup**.
5. Choose the restore mode (**New resource** or **Replace existing**).
6. Configure settings as needed and choose **Restore**.

Deleting backups

```
aws timestream-influxdb delete-db-backup \
    --identifier my-db-backup-id
```

If `retentionDays` is set, the DbBackup is automatically deleted on that day.

Retaining backups when deleting a resource

When deleting a resource, by default all automated DbBackups are deleted and on-demand backups are retained. To retain automated backups on resource deletion, use the `--retain-automated-backups` flag:

```
aws timestream-influxdb delete-db-instance \
    --identifier my-db-instance-id \
    --retain-automated-backups
```

```
aws timestream-influxdb delete-db-cluster \
    --db-cluster-id my-db-cluster-id \
    --retain-automated-backups
```

Retained automated backups are automatically deleted at their configured expiry day depending on the retention policy.

Performance impact

Taking a backup has minimal impact to the performance of the database. However, if the database is under high CPU and memory load, the backup may take longer to complete. It is recommended to schedule backups during low-traffic hours.

The first backup may take longer since it creates a full database backup. Subsequent backups are incremental and only capture changed data.

Relationship to service-managed backups

Customer-managed backups **complement** (not replace) the service-managed backups that AWS maintains:

Feature	Service-managed backups	Customer-managed backups
Control	Managed by AWS	Managed by you
Access	Via AWS support ticket	Self-service (Console, CLI, API)
Frequency	Hourly (automatic)	On-demand or custom schedule
Retention	Managed by AWS	Defined by you
Restore method	Support ticket required	Self-service restore (new resource or replace existing)

Both backup mechanisms can be active simultaneously, providing multiple layers of data protection.

Security and encryption

- All backups are encrypted at rest using service-managed AWS KMS keys.
- For resources with customer managed keys (CMK) enabled, backups are encrypted with the same AWS KMS key.
- Access to backup operations is controlled through IAM policies.
- Backup data remains in the same AWS Region as your database.

Limitations

- Backups can only be restored within the same AWS account and Region where they were created.
- Cross-Region and cross-account restore is not supported.
- A backup of a DB cluster cannot be restored to a standalone DB instance.
- Restoring a standalone instance (Single-AZ or Multi-AZ) backup to a Read Replicas cluster only includes data persisted to TSM files. See the section called “Restoring a standalone instance backup to a Read Replicas cluster”.
- A maximum of 4 DbBackupConfigurations can be enabled per resource.
- Custom schedule frequency must be at a daily frequency or less.

Pricing

There is no additional charge for the customer-managed backup and restore feature. You pay only for the Amazon EBS storage used by your backups. Incremental backups reduce storage costs by capturing only changed data. Restores are charged based on volume size. For Amazon EBS storage pricing, see Amazon EBS pricing.

Adding tags and labels to resources

You can label Amazon Timestream for InfluxDB resources using *tags*. Tags let you categorize your resources in different ways—for example, by purpose, owner, environment, or other criteria. Tags can help you do the following:

- Quickly identify a resource based on the tags that you assigned to it.
- See AWS bills broken down by tags.

Tagging is supported by AWS services like Amazon Elastic Compute Cloud (Amazon EC2), Amazon Simple Storage Service (Amazon S3), Timestream for InfluxDB, and more. Efficient tagging can provide cost insights by enabling you to create reports across services that carry a specific tag.

Finally, it is good practice to follow optimal tagging strategies. For information, see AWS Tagging Strategies.

Tagging restrictions

Each tag consists of a key and a value, both of which you define. The following restrictions apply:

- Each Timestream for InfluxDB DB instance can have only one tag with the same key. If you try to add an existing tag, the existing tag value is updated to the new value.
- A value acts as a descriptor within a tag category. In Timestream for InfluxDB the value cannot be empty or null.
- Tag keys and values are case sensitive.
- The maximum key length is 128 Unicode characters.
- The maximum value length is 256 Unicode characters.
- The allowed characters are letters, white space, and numbers, plus the following special characters: `+ - = . _ : /`
- The maximum number of tags per resource is 50.

- AWS-assigned tag names and values are automatically assigned the `aws:` prefix, which you can't assign. AWS-assigned tag names don't count toward the tag limit of 50. User-assigned tag names have the prefix `user:` in the cost allocation report.
- You can't backdate the application of a tag.

Security best practices for Timestream for InfluxDB

Optimize writes to InfluxDB

As any other time series database, InfluxDB is built to be able to ingest and process data in real-time. To keep the system performing at its best we recommend following optimizations when writing data to InfluxDB:

- **Batch Writes:** When writing data to InfluxDB, write data in batches to minimize the network overhead related to every write request. The optimal batch size is 5000 lines of line protocol per write request. To write multiple lines in one request, each line of line protocol must be delimited by a new line (\n).
- **Sort tags by key:** Before writing data points to InfluxDB, sort tags by key in lexicographic order.

```
measurement,tagC=therefore,tagE=am,tagA=i,tagD=i,tagB=think fieldKey=fieldValue
 1562020262

# Optimized line protocol example with tags sorted by key
measurement,tagA=i,tagB=think,tagC=therefore,tagD=i,tagE=am fieldKey=fieldValue
 1562020262
```

- **Use the coarsest time precision possible:** – InfluxDB writes data in nanosecond precision, however if your data isn’t collected in nanoseconds, there is no need to write at that precision. For better performance, use the coarsest precision possible for timestamps. You can specify the write precision when:
  - When using the SDK you can specify the WritePrecision when setting the time attribute of your point. For more information on InfluxDB client libraries, see the InfluxDB Documentation.
  - When using Telegraf, you configure the time precision in the Telegraf agent configuration. Precision is specified as an interval with an integer + unit (e.g. 0s,10ms,2us,4s). Valid time units are “ns”, “us”, “ms”, and “s”.

```
[agent]
```

```
interval ="10s"
metric_batch_size="5000"
precision = "0s"
```

- **Use gzip compression:** – Use gzip compression to speed up writes to InfluxDB and reduce network bandwidth. Benchmarks have shown up to a 5x speed improvement when data is compressed.
  - When using Telegraf, in the Influxdb_v2 output plugin configuration in your telegraf.conf, set the content_encoding option to gzip:

    ```
    [[outputs.influxdb_v2]]
      urls = ["http://localhost:8086"]
      # ...
      content_encoding = "gzip"
    ```

  - When using client libraries, each InfluxDB client library provides options for compressing write requests or enforces compression by default. The method for enabling compression is different for each library. For specific instructions, see the InfluxDB Documentation
  - When using the InfluxDB API `/api/v2/write` endpoint to write data, compress the data with gzip and set the Content-Encoding header to gzip.

Design for performance

Design your schema for simpler and more performance queries. The following guidelines will ensure that your schema will be easy to query and maximize query performance:

- **Design to query:** Choose measurements, tag keys, and field keys that are easy to query. To achieve this goal, follow these principles:
  - Use measurements that have a simple name and accurately describe the schema.
  - Avoid using the same name for a tag key and field key within the same schema.
  - Avoid using reserved Flux keywords and special characters in tag and field keys.
  - Tags store metadata that describe the fields and are common across many data points.
  - Fields store unique or highly variable data, usually numeric data points.
  - Measurements and keys should not contain data, but used to either aggregate or describe data. Data will be stored in tag and field values.
- **Keep your time series cardinality under control** High series cardinality is one of the main causes of decreased write and read performance in InfluxDB. In the context of InfluxDB high cardinality

refers to the presence of a very large number of unique tag values. Tags values are indexed in InfluxDB which means that a very high number of unique values will generate a larger index which can slow down data ingestion and query performance.

To better understand and resolve potential high cardinality related issues you can follow these steps:

- Understand the causes of high cardinality
- Measure the cardinality of your buckets
- Take action to resolve high cardinality

- **Causes of high series cardinality** InfluxDB indexes the data based on measurements and tags to speed up data reads. Each set of indexed data elements forms a series key. Tags containing highly variable information like unique IDs, hashes, and random strings lead to a large number of series, also known as high series cardinality. High series cardinality is the primary driver of high memory usage in InfluxDB.
- **Measuring series cardinality** If you experience performance slowdowns or see an ever increasing memory usage in your Timestream for InfluxDB instance, we recommend measureing the series cardinality of your buckets.

  InfluxDB provides functions that allows you to measure series cardinality both in Flux and InfluxQL.

  - In Flux use the function `influxdb.cardinality()`
  - In FluxQL use the `SHOW SERIES CARDINALITY` command

  In both cases the engine will return the number of unique series keys in your data. Keep in mind that is it not recommended to have more than 10 million series keys on any of your Timestream for InfluxDB instances.
- **Causes of high series cardinality** If you encounter that any of your buckets have high cardinality there are a few correcting steps you can take to fix it:
  - **Review your tags:** Ensure that your workloads don't generate cases were tags have unique values for most entries. This could happen in cases where the number of unique tag values always grows over time, or if log type messages are being written to the database where every message would have an unique combination of timestamp, tags etc. You can use the following Flux code to help you figure out which Tags are contributing most to your high cardinality issues:

```
// Count unique values for each tag in a bucketimport "influxdata/influxdb/schema"
```

```
cardinalityByTag = (bucket) => schema.tagKeys(bucket: bucket)
    |> map(
        fn: (r) => ({
            tag: r._value,
            _value: if contains(set: ["_stop", "_start"], value: r._value) then
                0
            else
                (schema.tagValues(bucket: bucket, tag: r._value)
                    |> count()
                    |> findRecord(fn: (key) => true, idx: 0))._value,
        }),
    )
    |> group(columns: ["tag"])
    |> sum()

cardinalityByTag(bucket: "amzn-s3-demo-bucket")
```

If you're experiencing very high cardinality, the query above may time out. If you experience a timeout, run the queries below – one at a time.

Generate a list of tags:

```
// Generate a list of tagsimport "influxdata/influxdb/schema"

schema.tagKeys(bucket: "amzn-s3-demo-bucket")
```

Count unique tag values for each tag:

```
// Run the following for each tag to count the number of unique tag valuesimport
 "influxdata/influxdb/schema"

tag = "example-tag-key"

schema.tagValues(bucket: "amzn-s3-demo-bucket1", tag: tag)
    |> count()
```

We recommend that you run these at different points in time to identify which tag is growing faster.

- **Improve your schema:** Follow the modeling recommendations discussed in our Security best practices for Timestream for InfluxDB.

- **Remove or aggregate older data to reduce cardinality:** Consider whether or not your use cases needs all the data that is causing your high cardinality issues. If this data is not longer needed or accessed frequently you can aggregate it, delete it or export it to another engine such as Timestream for Live Analytics for long term storage and analysis.

Troubleshooting

Warning of "dev" version not recognized

The warning 'WARN: Couldn't parse version "dev" reported by server, assuming latest backup/restore APIs are supported' may be displayed during migration. This warning can be ignored.

Migration failed during restoration stage

In the event of a failed migration during the restoration stage, users can use the `--retry-restore-dir` flag to re-attempt the restoration. Use the `--retry-restore-dir` flag with a path to a previously backed-up directory to skip the backup stage and retry the restoration stage. The created backup directory used for a migration will be indicated if a migration fails during restoration.

Possible reasons for a restore failing include:

- Invalid InfluxDB destination token – A bucket existing in the destination instance with the same name as in the source instance. For individual bucket migrations use the `--dest-bucket` option to set a unique name for the migrated bucket
- Connectivity failure, either with the source or destination hosts or with an optional S3 bucket.

Amazon Timestream for InfluxDB basic operational guidelines

Following are basic operational guidelines that everyone should follow when working with Amazon Timestream for InfluxDB. Note that the Amazon Timestream for InfluxDB Service Level Agreement requires that you follow these guidelines:

- Use metrics to monitor your memory, CPU, and storage usage. You can set up Amazon CloudWatch to notify you when usage patterns change or when you approach the capacity of your deployment. This way, you can maintain system performance and availability.

- Scale up your DB instance when you are approaching storage capacity limits. You should have some buffer in storage and memory to accommodate unforeseen increases in demand from your applications. Keep in mind that at this time, you will need to create a new instance and migrate your data to achieve this.
- If your database workload requires more I/O than you have provisioned, recovery after a failover or database failure will be slow. To increase the I/O capacity of a DB instance, do any or all of the following:
  - Migrate to a different DB instance with higher I/O capacity.
  - If you are already using Influx IOPS Included storage storage, provision a storage type with higher IOPS Included.
- If your client application is caching the Domain Name Service (DNS) data of your DB instances, set a time-to-live (TTL) value of less than 30 seconds. The underlying IP address of a DB instance can change after a failover. Caching the DNS data for an extended time can thus lead to connection failures. Your application might try to connect to an IP address that's no longer in service.

DB instance RAM recommendations

An Amazon Timestream for InfluxDB performance best practice is to allocate enough RAM so that your working set resides almost completely in memory. The working set is the data and indexes that are frequently in use on your instance. The more you use the DB instance, the more the working set will grow.

Amazon Timestream for InfluxDB event notifications with Amazon EventBridge

Amazon Timestream for InfluxDB publishes events to Amazon EventBridge when database instances or clusters undergo state changes. You can create EventBridge rules to route these events to any supported target to build automation, receive alerts, and maintain audit trails.

Overview

Amazon Timestream for InfluxDB generates events when operations such as instance creation, compute scaling, parameter group updates, maintenance, or deletion complete or fail. Events are published to the default EventBridge event bus in your account with source `aws.timestream-`

`influxdb`, supporting content-based filtering and routing to any EventBridge target including AWS Lambda functions, AWS Step Functions, Amazon Simple Queue Service queues, Amazon Simple Notification Service topics, and cross-account event buses.

To receive and act on these events, you create EventBridge rules that match the events you care about and route them to the targets of your choice. For example, you can:

- Route failure events to an Amazon SNS topic for on-call alerting via PagerDuty or email.
- Trigger a Lambda function when a compute scaling operation completes.
- Send all events to Amazon CloudWatch Logs for a compliance audit trail.
- Start a Step Functions workflow when a new cluster finishes creation.

There is no additional Amazon Timestream for InfluxDB charge for publishing events. Standard Amazon EventBridge pricing applies for rule evaluation and target delivery.

Event categories

Events are grouped into categories for use in EventBridge rule filtering:

Category	Description
`creation`	Instance or cluster created successfully, or creation failed
`notification`	Informational events for configuration changes, scaling, reboots, and other lifecycle operations
`maintenance`	Scheduled maintenance window started or ended
`failure`	All failure events across all operation types

> **Note**
>
> An event can belong to multiple categories. For example, a failed compute scaling event belongs to both `notification` and `failure`.

Source types

Source type	Description
DB_INSTANCE	InfluxDB v2 standalone instances
DB_CLUSTER	InfluxDB v2 Read Replica clusters and InfluxDB v3 clusters

Event schema

Events are published with the following structure:

```
{
  "version": "0",
  "id": "12345678-1234-1234-1234-123456789012",
  "detail-type": "Timestream InfluxDB DB Instance Event",
  "source": "aws.timestream-influxdb",
  "account": "123456789012",
  "time": "2026-06-15T14:30:00Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:timestream-influxdb:us-east-1:123456789012:db-instance/my-influxdb-
instance"
  ],
  "detail": {
    "EventID": "TIDB-EVENT-12001",
    "SourceType": "DB_INSTANCE",
    "SourceIdentifier": "my-influxdb-instance",
    "SourceArn": "arn:aws:timestream-influxdb:us-east-1:123456789012:db-instance/my-
influxdb-instance",
    "EventCategories": ["notification"],
    "Message": "InfluxDB v2 instance 'my-influxdb-instance' compute updated",
    "Date": "2026-06-15T14:30:00.000Z"
  }
}
```

The following example shows a failure event, which includes additional diagnostic fields:

```
{
  "version": "0",
  "id": "87654321-4321-4321-4321-210987654321",
```

```
  "detail-type": "Timestream InfluxDB DB Cluster Event",
  "source": "aws.timestream-influxdb",
  "account": "123456789012",
  "time": "2026-06-15T11:15:00Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:timestream-influxdb:us-east-1:123456789012:db-cluster/my-influxdb3-
cluster"
  ],
  "detail": {
    "EventID": "TIDB-EVENT-21101",
    "SourceType": "DB_CLUSTER",
    "SourceIdentifier": "my-influxdb3-cluster",
    "SourceArn": "arn:aws:timestream-influxdb:us-east-1:123456789012:db-cluster/my-
influxdb3-cluster",
    "EventCategories": ["creation", "failure"],
    "Message": "InfluxDB v3 cluster 'my-influxdb3-cluster' creation failed",
    "Date": "2026-06-15T11:15:00.000Z",
    "FailureCode": "INSUFFICIENT_EC2_CAPACITY",
    "FailureReason": "Insufficient capacity in requested Availability Zone",
    "RecommendedActions": "Retry the operation or try a different Availability Zone"
  }
}
```

Key fields:

- **source** — Always `aws.timestream-influxdb`
- **detail-type** — Either `Timestream InfluxDB DB Instance Event` or `Timestream InfluxDB DB Cluster Event`
- **detail.SourceType** — Either `DB_INSTANCE` or `DB_CLUSTER`
- **detail.EventCategories** — Array of categories this event belongs to (an event can belong to multiple categories)
- **detail.SourceIdentifier** — The name of the database instance or cluster
- **detail.Message** — Human-readable description including the InfluxDB engine version and resource name
- **detail.FailureCode** — (Failure events only) Machine-readable error code. Values include `INSUFFICIENT_EC2_CAPACITY`, `INSUFFICIENT_FREE_ADDRESSES_IN_SUBNET`, `INVALID_SUBNET_ID`, and `INTERNAL_ERROR`.
- **detail.FailureReason** — (Failure events only) Detailed description of the failure

- **detail.RecommendedActions** — (Failure events only) Suggested remediation steps as a string

Setting up event notifications

Step 1: Create an EventBridge rule

1. Open the Amazon EventBridge console.
2. Choose **Rules**, then **Create rule**.
3. Enter a rule name (for example, `timestream-influxdb-failures`).
4. For **Event bus**, keep **default**.
5. For **Rule type**, choose **Rule with an event pattern**.
6. Choose **Next**.

Step 2: Define the event pattern

Choose **Custom pattern** and enter a pattern that matches the events you want.

Match all Amazon Timestream for InfluxDB events:

```
{
    "source": ["aws.timestream-influxdb"],
    "detail-type": ["Timestream InfluxDB DB Instance Event", "Timestream InfluxDB DB
 Cluster Event"]
}
```

> **Note**
>
> Filtering by `detail-type` is recommended. Using only `"source": ["aws.timestream-influxdb"]` will also match CloudTrail events for the service, which can be noisy. The pattern above matches only Amazon Timestream for InfluxDB lifecycle events.

Match only failure events:

```
{
  "source": ["aws.timestream-influxdb"],
  "detail": {
    "EventCategories": ["failure"]
```

```
  }
}
```

Match events for a specific production cluster:

```
{
  "source": ["aws.timestream-influxdb"],
  "detail": {
    "SourceIdentifier": ["my-production-cluster"]
  }
}
```

Match maintenance and failure events for a specific resource:

```
{
  "source": ["aws.timestream-influxdb"],
  "detail": {
    "SourceIdentifier": ["my-production-cluster"],
    "EventCategories": ["maintenance", "failure"]
  }
}
```

Step 3: Select a target

Choose a target for matched events:

- **Amazon Simple Notification Service topic** — For email, SMS, or webhook notifications
- **AWS Lambda function** — For custom processing or remediation
- **Amazon Simple Queue Service queue** — For decoupled processing
- **AWS Step Functions** — For multi-step automation workflows
- **Amazon CloudWatch Logs** — For audit trail persistence

> **Important**
>
> Ensure your target's resource policy allows `events.amazonaws.com` to invoke it. For example, Amazon SNS topics must grant publish permission, Lambda functions must grant invoke permission, and Amazon SQS queues must grant send-message permission to `events.amazonaws.com`.

Step 4: Create the rule

Review and choose **Create rule**. Events matching your pattern will be routed to the target.

Common patterns

Email alerts on failures

1. Create an Amazon SNS topic (for example, `timestream-influxdb-alerts`) and subscribe your email address.
2. In the EventBridge console, create a rule matching `"EventCategories": ["failure"]`.
3. Set the Amazon SNS topic as the rule target.
4. Confirm the email subscription when you receive the confirmation message.

CI/CD pipeline integration

Create an EventBridge rule that triggers a Step Functions state machine when a specific operation completes:

```
{
  "source": ["aws.timestream-influxdb"],
  "detail": {
    "SourceIdentifier": ["my-staging-cluster"],
    "EventID": ["TIDB-EVENT-22001"]
  }
}
```

The state machine can then run integration tests, update DNS, and notify your deployment channel.

Maintenance window suppression

Create two EventBridge rules:

1. Rule matching `"EventID": ["TIDB-EVENT-24001"]` (maintenance started) → Lambda that sets a maintenance flag in your monitoring system.
2. Rule matching `"EventID": ["TIDB-EVENT-24002"]` (maintenance ended) → Lambda that clears the flag.

This prevents false-positive alerts during planned maintenance.

Audit trail with Amazon CloudWatch Logs

Create an EventBridge rule matching all events (`"source": ["aws.timestream-influxdb"]`) with a Amazon CloudWatch Logs log group as the target. Set the log group retention to your compliance requirement (for example, 1 year). All database lifecycle events are preserved for the retention period.

Instance event reference

Event ID	Operation	Category	Message
TIDB-EVENT-11001	DB_INSTANCE_CREATED	creation	InfluxDB *version* instance '*name*' created
TIDB-EVENT-11101	DB_INSTANCE_CREATE_FAILED	creation, failure	InfluxDB *version* instance '*name*' creation failed
TIDB-EVENT-13001	DB_INSTANCE_DELETED	notification	InfluxDB *version* instance '*name*' deleted
TIDB-EVENT-13101	DB_INSTANCE_DELETE_FAILED	notification, failure	InfluxDB *version* instance '*name*' deletion failed
TIDB-EVENT-16001	DB_INSTANCE_REBOOTED	notification	InfluxDB *version* instance '*name*' rebooted
TIDB-EVENT-16101	DB_INSTANCE_REBOOT_FAILED	notification, failure	InfluxDB *version* instance '*name*' reboot failed
TIDB-EVENT-12001	DB_INSTANCE_COMPUTE_UPDATED	notification	InfluxDB *version* instance '*name*' compute updated
TIDB-EVENT-12101	DB_INSTANCE_COMPUTE_UPDATE_FAILED	notification, failure	InfluxDB *version* instance '*name*' compute update failed

Event ID	Operation	Category	Message
TIDB-EVENT-12002	DB_INSTANCE_STORAGE_UPDATED	notification	InfluxDB *version* instance '*name*' storage updated
TIDB-EVENT-12102	DB_INSTANCE_STORAGE_UPDATE_FAILED	notification, failure	InfluxDB *version* instance '*name*' storage update failed
TIDB-EVENT-12003	DB_INSTANCE_PORT_UPDATED	notification	InfluxDB *version* instance '*name*' port updated
TIDB-EVENT-12103	DB_INSTANCE_PORT_UPDATE_FAILED	notification, failure	InfluxDB *version* instance '*name*' port update failed
TIDB-EVENT-12004	DB_INSTANCE_PARAMETER_GROUP_UPDATED	notification	InfluxDB *version* instance '*name*' parameter group updated
TIDB-EVENT-12104	DB_INSTANCE_PARAMETER_GROUP_UPDATE_FAILED	notification, failure	InfluxDB *version* instance '*name*' parameter group update failed
TIDB-EVENT-12005	DB_INSTANCE_LOG_DELIVERY_UPDATED	notification	InfluxDB *version* instance '*name*' log delivery updated
TIDB-EVENT-12105	DB_INSTANCE_LOG_DELIVERY_UPDATE_FAILED	notification, failure	InfluxDB *version* instance '*name*' log delivery update failed
TIDB-EVENT-12006	DB_INSTANCE_MAINTENANCE_WINDOW_UPDATED	notification	InfluxDB *version* instance '*name*' maintenance window updated
TIDB-EVENT-12106	DB_INSTANCE_MAINTENANCE_WINDOW_UPDATE_FAILED	notification, failure	InfluxDB *version* instance '*name*' maintenance window update failed
TIDB-EVENT-12009	DB_INSTANCE_UPDATED_TO_MAZ	notification	InfluxDB *version* instance '*name*' updated to Multi-AZ

Event ID	Operation	Category	Message
TIDB-EVENT-12109	DB_INSTANCE_UPDATE_TO_MAZ_FAILED	notification, failure	InfluxDB *version* instance '*name*' Multi-AZ update failed
TIDB-EVENT-12010	DB_INSTANCE_UPDATED_TO_SAZ	notification	InfluxDB *version* instance '*name*' updated to Single-AZ
TIDB-EVENT-12110	DB_INSTANCE_UPDATE_TO_SAZ_FAILED	notification, failure	InfluxDB *version* instance '*name*' Single-AZ update failed
TIDB-EVENT-14001	DB_INSTANCE_MAINTENANCE_WINDOW_STARTED	maintenance	InfluxDB *version* instance '*name*' maintenance started
TIDB-EVENT-14002	DB_INSTANCE_MAINTENANCE_WINDOW_ENDED	maintenance	InfluxDB *version* instance '*name*' maintenance ended

Cluster event reference

Event ID	Operation	Category	Message
TIDB-EVENT-21001	DB_CLUSTER_CREATED	creation	InfluxDB *version* cluster '*name*' created
TIDB-EVENT-21101	DB_CLUSTER_CREATE_FAILED	creation, failure	InfluxDB *version* cluster '*name*' creation failed
TIDB-EVENT-23001	DB_CLUSTER_DELETED	notification	InfluxDB *version* cluster '*name*' deleted
TIDB-EVENT-23101	DB_CLUSTER_DELETE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' deletion failed
TIDB-EVENT-26001	DB_CLUSTER_REBOOTED	notification	InfluxDB *version* cluster '*name*' rebooted

Event ID	Operation	Category	Message
TIDB-EVENT-26101	DB_CLUSTER_REBOOT_FAILED	notification, failure	InfluxDB *version* cluster '*name*' reboot failed
TIDB-EVENT-22001	DB_CLUSTER_COMPUTE_UPDATED	notification	InfluxDB *version* cluster '*name*' compute updated
TIDB-EVENT-22101	DB_CLUSTER_COMPUTE_UPDATE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' compute update failed
TIDB-EVENT-22002	DB_CLUSTER_STORAGE_UPDATED	notification	InfluxDB *version* cluster '*name*' storage updated
TIDB-EVENT-22102	DB_CLUSTER_STORAGE_UPDATE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' storage update failed
TIDB-EVENT-22003	DB_CLUSTER_PORT_UPDATED	notification	InfluxDB *version* cluster '*name*' port updated
TIDB-EVENT-22103	DB_CLUSTER_PORT_UPDATE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' port update failed
TIDB-EVENT-22004	DB_CLUSTER_PARAMETER_GROUP_UPDATED	notification	InfluxDB *version* cluster '*name*' parameter group updated
TIDB-EVENT-22104	DB_CLUSTER_PARAMETER_GROUP_UPDATE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' parameter group update failed
TIDB-EVENT-22005	DB_CLUSTER_LOG_DELIVERY_UPDATED	notification	InfluxDB *version* cluster '*name*' log delivery updated
TIDB-EVENT-22105	DB_CLUSTER_LOG_DELIVERY_UPDATE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' log delivery update failed

Event ID	Operation	Category	Message
TIDB-EVENT-22006	DB_CLUSTER_MAINTENANCE_WINDOW_UPDATED	notification	InfluxDB *version* cluster '*name*' maintenance window updated
TIDB-EVENT-22106	DB_CLUSTER_MAINTENANCE_WINDOW_UPDATE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' maintenance window update failed
TIDB-EVENT-22008	DB_CLUSTER_INSTANCE_MODES_UPDATED	notification	InfluxDB *version* cluster '*name*' instance modes updated
TIDB-EVENT-22108	DB_CLUSTER_INSTANCE_MODES_UPDATE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' instance modes update failed
TIDB-EVENT-22009	DB_CLUSTER_ENGINE_TYPE_CONVERTED	notification	InfluxDB *version* cluster '*name*' engine type converted
TIDB-EVENT-22109	DB_CLUSTER_ENGINE_TYPE_CONVERSION_FAILED	notification, failure	InfluxDB *version* cluster '*name*' engine type conversion failed
TIDB-EVENT-22011	DB_CLUSTER_INSTANCES_ADDED	notification	InfluxDB *version* cluster '*name*' instances added
TIDB-EVENT-22111	DB_CLUSTER_INSTANCE_ADD_FAILED	notification, failure	InfluxDB *version* cluster '*name*' instance addition failed
TIDB-EVENT-22012	DB_CLUSTER_INSTANCES_REMOVED	notification	InfluxDB *version* cluster '*name*' instances removed
TIDB-EVENT-22112	DB_CLUSTER_INSTANCE_REMOVE_FAILED	notification, failure	InfluxDB *version* cluster '*name*' instance removal failed
TIDB-EVENT-24001	DB_CLUSTER_MAINTENANCE_WINDOW_STARTED	maintenance	InfluxDB *version* cluster '*name*' maintenance started

Event ID	Operation	Category	Message
TIDB-EVENT-24002	DB_CLUSTER_MAINTENANCE_WINDOW_ENDED	maintenance	InfluxDB *version* cluster '*name*' maintenance ended

Troubleshooting

Events not appearing in EventBridge

Verify the operation has completed. Events are emitted only when a workflow reaches a terminal state (success or failure), not when it starts. Check your EventBridge rule pattern matches the event structure using the EventBridge console **Test event pattern** feature. Ensure you are looking at the correct Region—events are published in the same Region as the database resource.

Target not receiving events

Confirm your EventBridge rule is enabled and the target is correctly configured. Verify the target's resource policy allows `events.amazonaws.com` to invoke it. For Amazon SNS targets, ensure the topic policy grants publish permission and that your subscription is confirmed. Check CloudWatch metrics for the rule: `TriggeredRules`, `Invocations`, and `FailedInvocations`.

Events delayed

Amazon EventBridge provides at-least-once delivery with a 24-hour retry window. Under normal conditions, events are delivered shortly after operation completion. If your target is throttled or misconfigured, EventBridge retries delivery for up to 24 hours. Check CloudWatch metrics for the rule and verify your target is healthy.

Limitations

- Events are published to Amazon EventBridge only. There is no Amazon Timestream for InfluxDB-specific event subscription API or console page—use EventBridge rules to filter and route events.
- EventBridge does not retain events. To persist event history, route events to Amazon CloudWatch Logs, Amazon S3, or another durable store using an EventBridge rule. Historical events that occurred before a rule was created or modified will not be delivered—rules only match events from the point they are created or updated onwards.

- Events are emitted only for operations initiated through the Amazon Timestream for InfluxDB APIs (console, CLI, SDK, CloudFormation). Internal service operations do not generate customer-visible events.
- Events are regional—they are published in the same Region as the database resource.

Pricing

There is no additional Amazon Timestream for InfluxDB charge for publishing events. Standard pricing applies for downstream processing:

- Amazon EventBridge pricing — for rule evaluation and event delivery to targets
- Target service pricing applies based on which targets you configure (Lambda invocations, Amazon SQS messages, Amazon SNS deliveries, etc.)

Security in Timestream for InfluxDB

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to Timestream for InfluxDB, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using Timestream for InfluxDB. The following topics show you how to configure Timestream for InfluxDB to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your Timestream for InfluxDB resources.

Topics

- Overview
- Database authentication with Amazon Timestream for InfluxDB
- How Amazon Timestream for InfluxDB uses secrets
- Data protection in Timestream for InfluxDB
- Identity and Access Management for Amazon Timestream for InfluxDB
- Logging and monitoring in Timestream for InfluxDB
- Compliance validation for Amazon Timestream for InfluxDB
- Resilience in Amazon Timestream for InfluxDB
- Infrastructure security in Amazon Timestream for InfluxDB
- Configuration and vulnerability analysis in Timestream for InfluxDB
- Incident response in Timestream for InfluxDB
- Amazon Timestream for InfluxDB API and interface VPC endpoints (AWS PrivateLink)
- Security best practices for Timestream for InfluxDB

Overview

This documentation helps you understand how to apply the shared responsibility model when using Amazon Timestream for InfluxDB. The following topics show you how to configure Amazon Timestream for InfluxDB to meet your security and compliance objectives. You also learn how to use other AWS services that help you monitor and secure your Amazon Timestream for InfluxDB resources.

You can manage access to your Amazon Timestream for InfluxDB resources and your databases on a DB instance. The method you use to manage access depends on what type of task the user needs to perform with Amazon Timestream for InfluxDB:

- Run your DB instance in a Virtual Private Cloud (VPC) based on the Amazon VPC service for network access control.
- Use AWS Identity and Access Management (IAM) policies to assign permissions that determine who is allowed to manage Amazon Timestream for InfluxDB resources. For example, you can use IAM to determine who is allowed to create, describe, modify, and delete DB instances, tag resources, or modify security groups.

- Use security groups to control what IP addresses or Amazon EC2 instances can connect to your databases on a DB instance. When you first create a DB instance, it's only accessible through rules specified by an associated security group.
- Use Secure Socket Layer (SSL) or Transport Layer Security (TLS) connections with your DB instances.
- Use the security features of your InfluxDB engine to control who can log in to the databases on a DB instance. These features work just as if the database was on your local network. For more information, see Security in Timestream for InfluxDB.

> **Note**
>
> You have to configure security only for your use cases. You don't have to configure security access for processes that Amazon Timestream for InfluxDB manages. These include creating backups, replicating data between a primary DB instance and a read replica, and other processes.

Topics

- General security

General security

Topics

- Permissions
- Network access
- Dependencies
- S3 buckets

Permissions

InfluxDB users should be granted least-privilege permissions. Only tokens granted to specific users, instead of operator tokens, should be used during migration.

Timestream for InfluxDB uses IAM permissions to control user permissions. We recommend users be granted access to the specific actions and resources that they require. For more information, see Grant least privilege access.

Network access

The Influx migration script can function locally, migrating data between two InfluxDB instances on the same system, but it is assumed that the primary use case for migrations will be migrating data across the network, either a local or public network. With this comes security considerations. The Influx migration script will, by default, verify TLS certificates for instances with TLS enabled: we recommend that users enable TLS in their InfluxDB instances and do not use the `--skip-verify` option for the script.

We recommend you use an allow-list to restrict network traffic to be from sources you are expecting. You can do this by limiting network traffic to the InfluxDB instances only from known IPs.

Dependencies

The latest major versions of all dependencies should be used, including Influx CLI, InfluxDB, Python, the Requests module, and optional dependencies such as `mountpoint-s3` and `rclone`.

S3 buckets

If S3 buckets are used as a temporary storage for migration, we recommend enabling TLS, versioning, and disabling public access.

Using S3 buckets for migration

1. Open the AWS Management Console, navigate to **Amazon Simple Storage Service** and then choose **Buckets**.
2. Choose the bucket you wish to use.
3. Choose the **Permissions** tab.
4. Under **Block public access (bucket settings)**, choose **Edit**.
5. Check **Block all public access**.
6. Choose **Save changes**.
7. Under **Bucket policy**, choose **Edit**.
8. Enter the following, replacing *<example-bucket>* with your bucket name, to enforce the use of TLS version 1.2 or later for connections:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceTLSv12orHigher",
            "Principal": {
                "AWS": "*"
            },
            "Action": [
                "s3:*"
            ],
            "Effect": "Deny",
            "Resource": [
                "arn:aws:s3:::<example bucket>/*",
                "arn:aws:s3:::<example bucket>"
            ],
            "Condition": {
                "NumericLessThan": {
                    "s3:TlsVersion": 1.2
                }
            }
        }
    ]
}
```

9. Choose **Save changes**.
10. Choose the **Properties** tab.
11. Under **Bucket Versioning**, choose **Edit**.
12. Check **Enable**.
13. Choose **Save changes**.

For information about Amazon S3 bucket best security practices, see Security best practices for Amazon Simple Storage Service.

Database authentication with Amazon Timestream for InfluxDB

Amazon Timestream for InfluxDB supports two ways to authenticate database users.

Password and access Token database authentication use different methods of authenticating to the database. Therefore, a specific user can log in to a database using only one authentication method. In both cases InfluxDB performs all administration of user accounts and API tokens.

Password authentication

During the InfluxDB DB instance creation process, you created an organization, user and password. The user has permissions to manage everything in your Timestream for InfluxDB DB instance. With this username and password combination you will be able to LogIn into your instance using the InfluxUI and also use the InfluxCLI to generate an operator token.

An operator token is required to create users, delete buckets , organizations etc. For more information, see Database authentication options.

API tokens

InfluxDB API tokens ensure secure interaction between InfluxDB and external tools such as clients or applications. An API token belongs to a specific user and identifies InfluxDB permissions within the user's organization.

There are three types of API tokens in InfluxDB:

- Operator Token: Grants full read and write access to all organizations and all organization resources in InfluxDB OSS 2.x. Some operations, for example, retrieving the server configuration, require operator permissions. To create an operator token manually with the InfluxDB UI, `api/v2` API, or Influx CLI after the setup process is completed, you must use an existing operator token or your username and password. To create a new operator token without using an existing one, see the influxd recovery auth CLI.

  > **Important**
  >
  > Because operator tokens have full read and write access to all organizations in the database, we recommend creating an All-Access token for each organization and using those to manage InfluxDB. This helps to prevent accidental interactions across organizations.

- All-Access API Token: Grants full read and write access to all resources in an organization.
- Read/Write Tokens: Grants read access, write access, or both to specific buckets in an organization.

All InfluxDb tokens are long lived tokens with no set expiration date, so it is not recommended to use your operator or all access tokens to sent monitoring data from your clients or Telegraf agents neither to embed them in your dashboarding applications. For these applications create read/write tokens with just the necessary permissions to get the job done. Fo more information on how to create influxDB token, see Create a token.

Secrets

InfluxDB operator tokens are generated on instance setup; other kinds of tokens, such as all-access and read/write tokens, can be created using the Influx CLI, Influx v2 API, or the Timestream for InfluxDB Multi-user rotation function. See Manage API tokens for how to generate, view, assign, and delete tokens.

We recommend that you rotate Timestream for InfluxDB tokens often using AWS Secrets Manager and store tokens via environment variables. See Use Tokens for token usage in environment variables and Rotating the secret for how to rotate Timestream for InfluxDB users and tokens.

See also:

- Infrastructure security in Amazon Timestream for InfluxDB
- Security best practices for Timestream for InfluxDB

How Amazon Timestream for InfluxDB uses secrets

Timestream for InfluxDB supports username and password authentication through the user interface, and token credentials for least privilege client and application connections. Timestream for InfluxDB users have `allAccess` permissions within their organization while tokens can have any set of permissions. Following best practices for secure API token management, users should be created to manage tokens for fine-grain access within an organization. Additional information on admin best practices with Timestream for InfluxDB can be found in the Influxdata documentation.

AWS Secrets Manager is a secret storage service that you can use to protect database credentials, API keys, and other secret information. Then in your code, you can replace hardcoded credentials with an API call to Secrets Manager. This helps ensure that the secret can't be compromised by someone examining your code, because the secret isn't there. For an overview of Secrets Manager, see What is AWS Secrets Manager.

When you create a database instance, Timestream for InfluxDB automatically creates an admin secret for you to use with the multi-user rotation AWS Lambda function. In order to rotate

Timestream for InfluxDB users and tokens, you must create a new secret by hand for each user or token you wish to rotate. Each secret can be configured to rotate on a schedule with the use of a Lambda function. The process to setup a new rotating secret consists of uploading the Lambda function code, configuring the Lambda role, defining the new secret, and configuring the secret rotation schedule.

What's in the secret

When you store Timestream for InfluxDB user credentials in the secret, use the following format.

Single-user:

```
{
  "engine": "<required: must be set to 'timestream-influxdb'>",
  "username": "<required: username>",
  "password": "<required: password>",
  "dbIdentifier": "<required: DB identifier>"
}
```

When you create a Timestream for InfluxDB instance, an admin secret is automatically stored in Secrets Manager with credentials to be used with the multi-user Lambda function. Set the `adminSecretArn` to the `Authentication Properties Secret Manager ARN` value found on the DB instance summary page or to the ARN of an admin secret. To create a new admin secret you must already have the associated credentials and the credentials must have admin privileges.

When you store Timestream for InfluxDB token credentials in the secret, use the following format.

Multi-user:

```
{
  "engine": "<required: must be set to 'timestream-influxdb'>",
  "org": "<required: organization to associate token with>",
  "adminSecretArn": "<required: ARN of the admin secret>",
  "type": "<required: allAccess or operator or custom>",
  "dbIdentifier": "<required: DB identifier>",
  "token": "<required unless generating a new token: token being rotated>",
  "writeBuckets": "<optional: list of bucketIDs for custom type token, must be input
 within plaintext panel, for example ['id1','id2']>",
  "readBuckets": "<optional: list of bucketIDs for custom type token, must be input
 within plaintext panel, for example ['id1','id2']>",
  "permissions": "<optional: list of permissions for custom type token, must be input
 within plaintext panel, for example ['write-tasks','read-tasks']>"
```

```
}
```

When you store Timestream for InfluxDB admin credentials in the secret, use the following format:

Admin secret:

```
{
  "engine": "<required: must be set to 'timestream-influxdb'>",
  "username": "<required: username>",
  "password": "<required: password>",
  "dbIdentifier": "<required: DB identifier>",
  "organization": "<optional: initial organization>",
  "bucket": "<optional: initial bucket>"
}
```

To turn on automatic rotation for the secret, the secret must be in the correct JSON structure. See Rotating the secret for how to rotate Timestream for InfluxDB secrets.

Modifying the secret

The credentials generated during the Timestream for InfluxDB instance creation process are stored in a Secrets Manager secret in your account. The GetDbInstance response object contains an `influxAuthParametersSecretArn` which holds the Amazon Resource Name (ARN) to such secret. The secret will only be populated after your Timestream for InfluxDB instance is available. This is a READONLY copy as any updates/modifications/deletions to this secret doesn't impact the created DB instance. If you delete this secret, the API response will still refer to the deleted secret ARN.

To create a new token in the Timestream for InfluxDB instance rather than store existing token credentials, you can create non-operator tokens by leaving the `token` value blank in the secret and using the multi-user rotation function with the `AUTHENTICATION_CREATION_ENABLED` Lambda environment variable set to `true`. If you create a new token, the permissions defined in the secret are assigned to the token and cannot be altered after the first successful rotation. For more information on rotating secrets, see Rotating AWS Secrets Manager Secrets.

If a secret is deleted, the associated user or token in the Timestream for InfluxDB instance will not be deleted.

Rotating the secret

You use the Timestream for InfluxDB single- and multi-user rotation Lambda functions to rotate Timestream for InfluxDB user and token credentials. Use the single-user Lambda function to rotate user credentials for your Timestream for InfluxDB instance, and use the multi-user Lambda function to rotate token credentials for your Timestream for InfluxDB instance.

Rotating users and tokens with the single- and multi-user Lambda functions is optional. Timestream for InfluxDB credentials never expire and any exposed credentials pose a risk for malicious actions against your DB instance. The advantage of rotating Timestream for InfluxDB credentials with Secrets Manager is an added security layer which limits the attack vector of exposed credentials to the window of time until the next rotation cycle. If no rotation mechanism is in place for your DB instance, any exposed credentials will be valid until they are manually deleted.

You can configure Secrets Manager to automatically rotate secrets for you according to a schedule that you specify. This enables you to replace long-term secrets with short-term ones, which helps to significantly reduce the risk of compromise. For more information on rotating secrets with Secrets Manager, see Rotate AWS Secrets Manager Secrets.

Rotating users

When you rotate users with the single-user Lambda function, a new random password will be assigned to the user after each rotation. For more information on how to enable automatic rotation, see Set up automatic rotation for non-database AWS Secrets Manager secrets.

Rotating admin secrets

To rotate an admin secret you use the single-user rotation function. You need to add the `engine` and `dbIdentifier` values to the secret since those values are not automatically populated on DB initialization. See What's in the secret for the complete secret template.

To locate an admin secret for a Timestream for InfluxDB instance you use the admin secret ARN from the Timestream for InfluxDB instance summary page. It is recommended that you rotate all Timestream for InfluxDB admin secrets since admin users have elevated permissions for the Timestream for InfluxDB instance.

Lambda rotation function

You can rotate a Timestream for InfluxDB user with the single-user rotation function by using the What's in the secret with a new secret and adding the required fields for your Timestream for

InfluxDB user. For more information on secret rotation Lambda functions, see Rotation by Lambda function.

You can rotate a Timestream for InfluxDB user with the single-user rotation function by using the What's in the secret with a new secret and adding the required fields for your Timestream for InfluxDB user. For more information on secret rotation Lambda functions, see Rotation by Lambda function.

The single user rotation function authenticates with the Timestream for InfluxDB DB instance using the credentials defined in the secret, then generates a new random password and sets the new password for the user. For more information on secret rotation Lambda functions, see Rotation by Lambda function.

Lambda function execution role permissions

Use the following IAM policy as the role for the single-user Lambda function. The policy gives the Lambda function the required permissions to perform a secret rotation for Timestream for InfluxDB users.

Replace all items listed below in the IAM policy with values from your AWS account:

- **{rotating_secret_arn}** — The ARN for the secret being rotated can be found in the Secrets Manager secret details.
- **{db_instance_arn}** — The Timestream for InfluxDB instance ARN can be found on the Timestream for InfluxDB instance summary page.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:DescribeSecret",
                "secretsmanager:GetSecretValue",
                "secretsmanager:PutSecretValue",
                "secretsmanager:UpdateSecretVersionStage"
            ],
```

```
            "Resource": "arn:aws:secretsmanager:us-
east-2:111122223333:secret:MySecret"
        },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetRandomPassword"
            ],
            "Resource": "*"
        },
        {
            "Action": [
                "timestream-influxdb:GetDbInstance"
            ],
            "Resource": "arn:aws:timestream-influxdb:us-east-2:111122223333:db-
instance/MyDbInstance",
            "Effect": "Allow"
        }
    ]
}
```

Rotating tokens

You can rotate a Timestream for InfluxDB token with the multi-user rotation function by using the What's in the secret with a new secret and adding the required fields for your Timestream for InfluxDB token. For more information on secret rotation Lambda functions, see Rotation by Lambda function.

You can rotate a Timestream for InfluxDB token by using the Timestream for InfluxDB multi-user Lambda function. Set the `AUTHENTICATION_CREATION_ENABLED` environment variable to `true` in the Lambda configuration to enable token creation. To create a new token, use the What's in the secret for your secret value. Omit the `token` key-value pair in the new secret and set the `type` to `allAccess`, or define the specific permissions and set the type to `custom`. The rotation function will create a new token during the first rotation cycle. You can't change the token permissions by editing the secret after rotation and any subsequent rotations will use the permissions that are set in the DB instance.

Lambda rotation function

The multi-user rotation function rotates token credentials by creating a new permission identical token using the admin credentials in the admin secret. The Lambda function validates the token

value in the secret before creating the replacement token, storing the new token value in the secret, and deleting the old token. If the Lambda function is creating a new token it will first validate that the `AUTHENTICATION_CREATION_ENABLED` environment variable is set to `true`, that there is no token value in the secret, and that the token type is not type operator.

Lambda function execution role permissions

Use the following IAM policy as the role for the multi-user Lambda function. The policy gives the Lambda function the required permissions to perform a secret rotation for Timestream for InfluxDB tokens.

Replace all items listed below in the IAM policy with values from your AWS account:

- **{rotating_secret_arn}** — The ARN for the secret being rotated can be found in the Secrets Manager secret details.
- **{authentication_properties_admin_secret_arn}** — The Timestream for InfluxDB admin secret ARN can be found on the Timestream for InfluxDB instance summary page.
- **{db_instance_arn}** — The Timestream for InfluxDB instance ARN can be found on the Timestream for InfluxDB instance summary page.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:DescribeSecret",
                "secretsmanager:GetSecretValue",
                "secretsmanager:PutSecretValue",
                "secretsmanager:UpdateSecretVersionStage"
            ],
         "Resource": "arn:aws:secretsmanager:us-
east-2:111122223333:secret:MySecret"
        },
        {
            "Effect": "Allow",
            "Action": [
```

```
                "secretsmanager:GetSecretValue"
            ],
         "Resource": "arn:aws:secretsmanager:us-
east-2:111122223333:secret:MyAdminSecret"
        },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetRandomPassword"
            ],
            "Resource": "*"
        },
        {
            "Action": [
                "timestream-influxdb:GetDbInstance"
            ],
         "Resource": "arn:aws:timestream-influxdb:us-east-2:111122223333:db-
instance/MyDbInstance",
            "Effect": "Allow"
        }
    ]
}
```

Data protection in Timestream for InfluxDB

The AWS shared responsibility model applies to data protection in Amazon Timestream for InfluxDB. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.

- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Timestream for InfluxDB or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

For more detailed information on Timestream for InfluxDB data protection topics like Encryption at Rest and Key Management, select any of the available topics below.

Topics

- Encryption at rest
- Encrypting resources with customer managed keys
- Encryption in transit

Encryption at rest

Timestream for InfluxDB encryption at rest provides enhanced security by encrypting all your data at rest using encryption keys stored in AWS Key Management Service (AWS KMS). This functionality helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet strict encryption compliance and regulatory requirements.

- Encryption is turned on by default on your Timestream for InfluxDB DB instance, and cannot be turned off. The industry standard AES-256 encryption algorithm is the default encryption algorithm used.
- AWS KMS is used for encryption at rest in Timestream for InfluxDB.
- You don't need to modify your DB instance client applications to use encryption.

Encrypting resources with customer managed keys

Amazon Timestream for InfluxDB encrypts your data at rest by default using AWS owned encryption keys. You can optionally specify a customer managed key (CMK) when creating a new DB instance or cluster to control data encryption with your own AWS Key Management Service (AWS KMS) key.

Overview

When you specify a customer managed key during resource creation, Amazon Timestream for InfluxDB uses your key to encrypt the Amazon EBS volumes that store your database data.

Using a customer managed key gives you full control over the encryption key lifecycle, including the ability to create, rotate, disable, and define access policies. You can also audit key usage through AWS CloudTrail.

> **Important**
>
> You can only specify a customer managed key during resource creation. You cannot change the encryption key after an instance or cluster has been created.

Supported key types

Amazon Timestream for InfluxDB requires an AWS KMS symmetric encryption key (`SYMMETRIC_DEFAULT`). The key must meet these requirements:

- Key type: Symmetric (`SYMMETRIC_DEFAULT`)
- Key usage: Encrypt and decrypt
- Key origin: AWS KMS (customer managed)
- Key Region: Same AWS Region as the database instance or cluster

- Multi-Region keys: Supported (the primary or replica key must be in the same Region as the database)

Creating a database with a customer managed key

Using the AWS CLI

To create a DB instance with a customer managed key, specify the `--kms-key-id` parameter with the ARN of your AWS KMS key.

InfluxDB 2 instance:

```
aws timestream-influxdb create-db-instance \
    --name my-influxdb-instance \
    --db-instance-type db.influx.medium \
    --db-storage-type InfluxIOIncludedT1 \
    --allocated-storage 100 \
    --vpc-subnet-ids subnet-0123456789abcdef0 subnet-0123456789abcdef1 \
    --vpc-security-group-ids sg-0123456789abcdef0 \
    --password MySecurePassword123! \
    --kms-key-id arn:aws:kms:us-
east-1:123456789012:key/12345678-1234-1234-1234-123456789012
```

InfluxDB 2 Read Replicas cluster:

```
aws timestream-influxdb create-db-cluster \
    --name my-influxdb-cluster \
    --db-instance-type db.influx.medium \
    --db-storage-type InfluxIOIncludedT1 \
    --allocated-storage 100 \
    --vpc-subnet-ids subnet-0123456789abcdef0 subnet-0123456789abcdef1 \
    --vpc-security-group-ids sg-0123456789abcdef0 \
    --password MySecurePassword123! \
    --deployment-type MULTI_NODE_READ_REPLICAS \
    --failover-mode NO_FAILOVER \
    --kms-key-id arn:aws:kms:us-
east-1:123456789012:key/12345678-1234-1234-1234-123456789012
```

Using the

1. Open the Amazon Timestream for InfluxDB console.

2. Choose **Create database**.
3. In the **Encryption** section, select **Customer managed key**.
4. Choose your AWS KMS key from the dropdown list, or enter the key ARN.
5. Complete the remaining configuration and choose **Create**.

Setting up the KMS key policy

Before creating a database with a customer managed key, you must configure the key policy to grant Amazon Timestream for InfluxDB the necessary permissions. The key policy requires two statements: one to allow the calling role to create grants through the service, and one to allow the role to describe the key.

Add the following statements to your key policy, replacing *account-id*, *caller-role*, and *region* with your values:

```
{
    "Sid": "Allow Timestream InfluxDB to use the key for resource allocations",
    "Effect": "Allow",
    "Principal": {
        "AWS": "arn:aws:iam::account-id:role/caller-role"
    },
    "Action": "kms:CreateGrant",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "kms:ViaService": "timestream-influxdb.region.amazonaws.com"
        },
        "ForAllValues:StringEquals": {
            "kms:GrantOperations": [
                "Decrypt",
                "Encrypt",
                "GenerateDataKey",
                "GenerateDataKeyWithoutPlaintext",
                "ReEncryptFrom",
                "ReEncryptTo",
                "CreateGrant",
                "DescribeKey"
            ]
        },
        "Bool": {
            "kms:GrantIsForAWSResource": "true"
```

```
        }
    }
},
{
    "Sid": "Allow Timestream InfluxDB to describe the key for resource allocations",
    "Effect": "Allow",
    "Principal": {
        "AWS": "arn:aws:iam::account-id:role/caller-role"
    },
    "Action": "kms:DescribeKey",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "kms:ViaService": "timestream-influxdb.region.amazonaws.com"
        }
    }
}
```

The first statement allows the caller's IAM role to create grants through the Amazon Timestream for InfluxDB service, scoped to specific grant operations. The `kms:ViaService` condition ensures this permission can only be exercised through the service, and `kms:GrantIsForAWSResource` ensures grants are only created for AWS resources.

The second statement allows the caller's IAM role to describe the key through the service.

Viewing the encryption key

To view the AWS KMS key used by a DB instance or cluster, use the `GetDbInstance` or `GetDbCluster` API. The response includes the `kmsKeyId` field if a customer managed key was specified during creation.

```
aws timestream-influxdb get-db-instance \
    --db-instance-identifier my-influxdb-instance
```

If no customer managed key was specified, the `kmsKeyId` field is not present in the response, indicating that the database uses the default AWS owned key.

Monitoring key usage with CloudTrail

When you use a customer managed key, AWS CloudTrail records all AWS KMS API calls made by Amazon Timestream for InfluxDB. You can use these logs to audit key usage and monitor access patterns.

Event	When it occurs
`CreateGrant`	During instance or cluster creation, to allow Amazon EBS to use the key
`GenerateDataKeyWithoutPlaintext`	When Amazon EBS creates encrypted volumes
`Decrypt`	When Amazon EBS reads data from encrypted volumes

The `invokedBy` field in these events shows `timestream-influxdb.`*`region`*`.amazonaws.com`, identifying the service that initiated the request.

Key revocation and deletion

If you disable or delete a customer managed key, or revoke the grants, Amazon Timestream for InfluxDB loses access to the encrypted data. This results in the following behavior:

- The database instance or cluster becomes unavailable.
- Read and write operations fail.
- The database remains in a degraded state until key access is restored.

If you re-enable the key or restore the grants, the database automatically recovers and becomes operational again.

> **Warning**
>
> If you permanently delete a customer managed key (after the AWS KMS waiting period), the encrypted data is permanently unrecoverable. Always ensure that you no longer need access to the data before scheduling key deletion.

Service-managed backups: Service-managed backups (Amazon EBS snapshots) are encrypted with the same customer managed key. If the key is deleted, backups cannot be restored.

Migrating existing databases to use a customer managed key

Because you cannot change the encryption key after database creation, migrating an existing database to use a customer managed key requires:

1. Create a new DB instance or cluster with the `--kms-key-id` parameter.
2. Migrate data from the existing database to the new database using the `influx backup` and `influx restore` CLI commands.
3. Update your application connection settings to point to the new database.
4. Delete the old database when migration is complete.

Limitations

- Customer managed keys can only be specified during resource creation—not updated afterward.
- The AWS KMS key must be in the same AWS Region and account as the database.
- Only symmetric encryption keys (`SYMMETRIC_DEFAULT`) are supported.
- Root volumes (operating system) are not encrypted with the customer managed key—they continue to use service-managed encryption.

Pricing

There is no additional Amazon Timestream for InfluxDB charge for using customer managed keys. Standard AWS KMS pricing applies for your key and the API calls made by the service. For details, see AWS KMS pricing.

Encryption in transit

All your Timestream for InfluxDB data is encrypted in transit. By default, all communications to and from Timestream for InfluxDB are protected by using Transport Layer Security (TLS) encryption.

Traffic to and from Amazon Timestream for InfluxDB is secured using supported TLS versions 1.2 or 1.3.

Identity and Access Management for Amazon Timestream for InfluxDB

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in)

and *authorized* (have permissions) to use Timestream for InfluxDB resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Authenticating with identities
- Managing access using policies
- How Amazon Timestream for InfluxDB works with IAM
- Identity-based policy examples for Amazon Timestream for InfluxDB
- Troubleshooting Amazon Timestream for InfluxDB identity and access
- Controlling access to a DB instance in a VPC
- Using service-linked roles for Amazon Timestream for InfluxDB
- AWS managed policies for Amazon Timestream for InfluxDB
- Connecting to Timestream for InfluxDB through a VPC endpoint

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-

based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Timestream for InfluxDB works with IAM

IAM features you can use with Amazon Timestream for InfluxDB

IAM feature	Timestream for InfluxDB support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	No
ACLs	No
ABAC (tags in policies)	Yes
Temporary credentials	Yes
Principal permissions	Yes
Service roles	No
Service-linked roles	Yes

To get a high-level view of how Timestream for InfluxDB and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Timestream for InfluxDB

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Timestream for InfluxDB

To view examples of Timestream for InfluxDB identity-based policies, see Identity-based policy examples for Amazon Timestream for InfluxDB.

Resource-based policies within Timestream for InfluxDB

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for Timestream for InfluxDB

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Timestream for InfluxDB actions, see Actions, resources and condition keys for Amazon Timestream for InfluxDB in the *Service Authorization Reference*.

Policy actions in Timestream for InfluxDB use the following prefix before the action:

```
timestream-influxdb
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "timestream-influxdb:action1",
      "timestream-influxdb:action2"
         ]
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Describe`, include the following action:

```
"Action": "timestream-influxdb:Describe*"
```

Policy resources for Timestream for InfluxDB

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Timestream for InfluxDB resource types and their ARNs, see Resource types defined by Amazon Timestream for InfluxDB in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions, resources and condition keys for Amazon Timestream for InfluxDB.

Policy condition keys for Timestream for InfluxDB

Supports service-specific policy condition keys: No

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

Access control lists (ACLs) in Timestream for InfluxDB

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) with Timestream for InfluxDB

Supports ABAC (tags in policies): Yes

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes called tags. You can attach tags to IAM entities and AWS resources, then design ABAC policies to allow operations when the principal's tag matches the tag on the resource.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using Temporary credentials with Timestream for InfluxDB

Supports temporary credentials: Yes

Temporary credentials provide short-term access to AWS resources and are automatically created when you use federation or switch roles. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM and AWS services that work with IAM in the *IAM User Guide*.

Cross-service principal permissions for Timestream for InfluxDB

Supports forward access sessions (FAS): Yes

Forward access sessions (FAS) use the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. For policy details when making FAS requests, see Forward access sessions.

Service roles for Timestream for InfluxDB

Supports service roles: No

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break Timestream for InfluxDB functionality. Edit service roles only when Timestream for InfluxDB provides guidance to do so.

Service-linked roles for Timestream for InfluxDB

Supports service-linked roles: Yes

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Amazon Timestream for InfluxDB

By default, users and roles don't have permission to create or modify Timestream for InfluxDB resources. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Timestream for InfluxDB, including the format of the ARNs for each of the resource types, see Actions, resources, and condition Keys for Amazon Timestream for InfluxDB in the *Service Authorization Reference*.

Topics

- Policy best practices
- Using the Timestream for InfluxDB console
- Allow users to view their own permissions
- Accessing one Amazon S3 bucket
- Allowing all operations
- Create, describe, delete and update a DB instance

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Timestream for InfluxDB resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more

information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.

- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Timestream for InfluxDB console

To access the Amazon Timestream for InfluxDB console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Timestream for InfluxDB resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Timestream for InfluxDB console, also attach the Timestream for InfluxDB `ConsoleAccess` or `ReadOnly` AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Accessing one Amazon S3 bucket

In this example, you want to grant an IAM user in your AWS account access to one of your Amazon S3 buckets, `amzn-s3-demo-bucket`. You also want to allow the user to add, update, and delete objects.

In addition to granting the `s3:PutObject`, `s3:GetObject`, and `s3:DeleteObject` permissions to the user, the policy also grants the `s3:ListAllMyBuckets`, `s3:GetBucketLocation`, and `s3:ListBucket` permissions. These are the additional permissions required by the console. Also, the `s3:PutObjectAcl` and the `s3:GetObjectAcl` actions are required to be able to copy, cut, and paste objects in the console. For an example walkthrough that grants permissions to users and tests them using the console, see [An example walkthrough: Using user policies to control access to your bucket](#).

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"ListBucketsInConsole",
         "Effect":"Allow",
         "Action":[
            "s3:ListAllMyBuckets"
         ],
         "Resource":"arn:aws:s3:::*"
      },
      {
         "Sid":"ViewSpecificBucketInfo",
         "Effect":"Allow",
         "Action":[
            "s3:ListBucket",
            "s3:GetBucketLocation"
         ],
         "Resource":"arn:aws:s3:::amzn-s3-demo-bucket"
      },
      {
         "Sid":"ManageBucketContents",
         "Effect":"Allow",
         "Action":[
            "s3:PutObject",
```

```
            "s3:PutObjectAcl",
            "s3:GetObject",
            "s3:GetObjectAcl",
            "s3:DeleteObject"
        ],
        "Resource":"arn:aws:s3:::amzn-s3-demo-bucket/*"
    }
  ]
}
```

Allowing all operations

The following is a sample policy that allows all operations in Timestream for InfluxDB.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream-influxdb:*"
            ],
            "Resource": "*"
        }
    ]
}
```

Create, describe, delete and update a DB instance

The following sample policy allows a user to create, describe, delete and update a DB instance `sampleDB`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "timestream-influxdb:CreateDbInstance",
                "timestream-influxdb:GetDbInstance",
                "timestream-influxdb:DeleteDbInstance",
                "timestream-influxdb:UpdateDbInstance"
            ],
            "Resource": "arn:aws:timestream-influxdb:us-east-2:111122223333:db-
instance/MyDbInstance"
        }
    ]
}
```

Troubleshooting Amazon Timestream for InfluxDB identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Timestream for InfluxDB and IAM.

Topics

- I am not authorized to perform an action in Timestream for InfluxDB
- I want to allow people outside of my AWS account to access my Timestream for InfluxDB resources

I am not authorized to perform an action in Timestream for InfluxDB

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the `mateojackson` user tries to use the console to view details about a fictional *`my-example-widget`* resource but does not have the fictional `timestream-influxdb:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 timestream-influxdb:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the `my-example-widget` resource using the `timestream-influxdb:GetWidget` action.

I want to allow people outside of my AWS account to access my Timestream for InfluxDB resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- Controlling access to a DB instance in a VPC
- To learn whether Timestream for InfluxDB supports these features, see How Amazon Timestream for InfluxDB works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.

Controlling access to a DB instance in a VPC

Using Amazon Virtual Private Cloud (Amazon VPC), you can launch AWS resources, such as Amazon Timestream for InfluxDB DB instances, into a virtual private cloud (VPC). When you use Amazon VPC, you have control over your virtual networking environment. You can choose your own IP address range, create subnets, and configure routing and access control lists.

A VPC security group controls access to DB instances inside a VPC. Each VPC security group rule enables a specific source to access a DB instance in a VPC that is associated with that VPC security group. The source can be a range of addresses (for example, 203.0.113.0/24), or another VPC security group. By specifying a VPC security group as the source, you allow incoming traffic from all instances (typically application servers) that use the source VPC security group. Before attempting

to connect to your DB instance, configure your VPC for your use case. The following are common scenarios for accessing a DB instance in a VPC:

A DB instance in a VPC accessed by an Amazon EC2 instance in the same VPC

A common use of a DB instance in a VPC is to share data with an application server that is running in an EC2 instance in the same VPC. The EC2 instance might run a web server with an application that interacts with the DB instance.

A DB instance in a VPC accessed by an EC2 instance in a different VPC

In some cases, your DB instance is in a different VPC from the EC2 instance that you're using to access it. If so, you can use VPC peering to access the DB instance.

A DB instance in a VPC accessed by a client application through the internet

To access a DB instance in a VPC from a client application through the internet, you configure a VPC with a single public subnet and use the public subnets to create the DB instance. You also configure an internet gateway in the VPC to enable communication over the internet. To connect to a DB instance from outside of its VPC, the DB instance must be publicly accessible. Also, access must be granted using the inbound rules of the DB instance's security group, and other requirements must be met.

For more information on VPC security groups, see Control traffic to your AWS resources using security groups in the *Amazon Virtual Private Cloud User Guide*.

For details on how to connect to a Timestream for InfluxDB DB instance, see Connecting to an Amazon Timestream for InfluxDB DB instance.

Security group scenario

A common use of a DB instance in a VPC is to share data with an application server running in an Amazon EC2 instance in the same VPC, which is accessed by a client application outside the VPC. For this scenario, you use the Timestream for InfluxDB and VPC pages on the AWS Management Console or the Timestream for InfluxDB and EC2 API operations to create the necessary instances and security groups:

1. Create a VPC security group (for example, `sg-0123ec2example`) and define inbound rules that use the IP addresses of the client application as the source. This security group allows your client application to connect to EC2 instances in a VPC that uses this security group.

2. Create an EC2 instance for the application and add the EC2 instance to the VPC security group (`sg-0123ec2example`) that you created in the previous step.
3. Create a second VPC security group (for example, `sg-6789rdsexample`) and create a new rule by specifying the VPC security group that you created in step 1 (`sg-0123ec2example`) as the source.
4. Create a new DB instance and add the DB instance to the VPC security group (`sg-6789rdsexample`) that you created in the previous step. When you create the DB, use the same port number as the one specified for the VPC security group (`sg-6789rdsexample`) rule that you created in step 3.

Creating a VPC security group

You can create a VPC security group for a DB instance by using the VPC console. For information about creating a security group, see Create a security group for your VPC in the *Amazon Virtual Private Cloud User Guide*.

Associating a security group with a DB instance

Once a Timestream for InfluxDB DB instance has been created, you will not be able to associate it to new security groups since changes to these configurations are not currently supported.

Using service-linked roles for Amazon Timestream for InfluxDB

Amazon Timestream for InfluxDB uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to an AWS service, such as Amazon Timestream for InfluxDB. Amazon Timestream for InfluxDB service-linked roles are predefined by Amazon Timestream for InfluxDB. They include all the permissions that the service requires to call AWS services on behalf of your dbinstances.

A service-linked role makes setting up Amazon Timestream for InfluxDB easier because you don’t have to manually add the necessary permissions. The roles already exist within your AWS account but are linked to Amazon Timestream for InfluxDB use cases and have predefined permissions. Only Amazon Timestream for InfluxDB can assume these roles, and only these roles can use the predefined permissions policy. You can delete the roles only after first deleting their related resources. This protects your Amazon Timestream for InfluxDB resources because you can't inadvertently remove necessary permissions to access the resources.

For information about other services that support service-linked roles, see AWS services that work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Contents

- Service-Linked Role Permissions for Amazon Timestream for InfluxDB
- Creating a Service-Linked Role (IAM)
- Editing the Description of a Service-Linked Role for Amazon Timestream for InfluxDB
  - Editing a Service-Linked Role Description (IAM Console)
  - Editing a Service-Linked Role Description (IAM CLI)
  - Editing a Service-Linked Role Description (IAM API)
- Deleting a Service-Linked Role for Amazon Timestream for InfluxDB
  - Cleaning Up a Service-Linked Role
  - Deleting a Service-Linked Role (IAM Console)
  - Deleting a Service-Linked Role (IAM CLI)
  - Deleting a Service-Linked Role (IAM API)
- Supported Regions for Amazon Timestream for InfluxDB Service-Linked Roles

Service-Linked Role Permissions for Amazon Timestream for InfluxDB

Amazon Timestream for InfluxDB uses the service-linked role named **AmazonTimestreamInfluxDBServiceRolePolicy** – This policy allows Timestream for InfluxDB to manage AWS resources on your behalf as necessary for managing your clusters.

The AmazonTimestreamInfluxDBServiceRolePolicy service-linked role permissions policy allows Amazon Timestream for InfluxDB to complete the following actions on the specified resources:

JSON

```
{
 "Version":"2012-10-17",
 "Statement": [
  {
   "Sid": "DescribeNetworkStatement",
   "Effect": "Allow",
   "Action": [
```

```
    "ec2:DescribeSubnets",
    "ec2:DescribeVpcs",
    "ec2:DescribeNetworkInterfaces"
   ],
   "Resource": "*"
  },
  {
   "Sid": "CreateEniInSubnetStatement",
   "Effect": "Allow",
   "Action": [
    "ec2:CreateNetworkInterface"
   ],
   "Resource": [
    "arn:aws:ec2:*:*:subnet/*",
    "arn:aws:ec2:*:*:security-group/*"
   ]
  },
  {
   "Sid": "CreateEniStatement",
   "Effect": "Allow",
   "Action": [
    "ec2:CreateNetworkInterface"
   ],
   "Resource": "arn:aws:ec2:*:*:network-interface/*",
   "Condition": {
    "Null": {
     "aws:RequestTag/AmazonTimestreamInfluxDBManaged": "false"
    }
   }
  },
  {
   "Sid": "CreateTagWithEniStatement",
   "Effect": "Allow",
   "Action": [
    "ec2:CreateTags"
   ],
   "Resource": "arn:aws:ec2:*:*:network-interface/*",
   "Condition": {
    "Null": {
     "aws:RequestTag/AmazonTimestreamInfluxDBManaged": "false"
    },
    "StringEquals": {
     "ec2:CreateAction": [
      "CreateNetworkInterface"
```

```
      ]
     }
    }
   },
   {
    "Sid": "ManageEniStatement",
    "Effect": "Allow",
    "Action": [
     "ec2:CreateNetworkInterfacePermission",
     "ec2:DeleteNetworkInterface"
    ],
    "Resource": "arn:aws:ec2:*:*:network-interface/*",
    "Condition": {
     "Null": {
      "aws:ResourceTag/AmazonTimestreamInfluxDBManaged": "false"
     }
    }
   },
   {
    "Sid": "PutCloudWatchMetricsStatement",
    "Effect": "Allow",
    "Action": [
     "cloudwatch:PutMetricData"
    ],
    "Condition": {
     "StringEquals": {
      "cloudwatch:namespace": [
       "AWS/Timestream/InfluxDB",
       "AWS/Usage"
      ]
     }
    },
    "Resource": [
     "*"
    ]
   },
   {
    "Sid": "ManageSecretStatement",
    "Effect": "Allow",
    "Action": [
     "secretsmanager:CreateSecret",
     "secretsmanager:DeleteSecret"
    ],
    "Resource": [
```

```
      "arn:aws:secretsmanager:*:*:secret:READONLY-InfluxDB-auth-parameters-*"
     ],
     "Condition": {
      "StringEquals": {
       "aws:ResourceAccount": "${aws:PrincipalAccount}"
      }
     }
    }
   ]
  }
```

To allow an IAM entity to create AmazonTimestreamInfluxDBServiceRolePolicy service-linked roles

Add the following policy statement to the permissions for that IAM entity:

```
{
    "Effect": "Allow",
    "Action": [
        "iam:CreateServiceLinkedRole",
        "iam:PutRolePolicy"
    ],
    "Resource": "arn:aws:iam::*:role/aws-service-role/
timestreamforinfluxdb.amazonaws.com/AmazonTimestreamInfluxDBServiceRolePolicy*",
    "Condition": {"StringLike": {"iam:AWSServiceName":
 "timestreamforinfluxdb.amazonaws.com"}}
}
```

To allow an IAM entity to delete AmazonTimestreamInfluxDBServiceRolePolicy service-linked roles

Add the following policy statement to the permissions for that IAM entity:

```
{
    "Effect": "Allow",
    "Action": [
        "iam:DeleteServiceLinkedRole",
        "iam:GetServiceLinkedRoleDeletionStatus"
    ],
    "Resource": "arn:aws:iam::*:role/aws-service-role/
timestreamforinfluxdb.amazonaws.com/AmazonTimestreamInfluxDBServiceRolePolicy*",
```

```
    "Condition": {"StringLike": {"iam:AWSServiceName":
 "timestreamforinfluxdb.amazonaws.com"}}
}
```

Alternatively, you can use an AWS managed policy to provide full access to Amazon Timestream for InfluxDB.

Creating a Service-Linked Role (IAM)

You don't need to manually create a service-linked role. When you create a DB instance, Amazon Timestream for InfluxDB creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you create a DB instance, Amazon Timestream for InfluxDB creates the service-linked role for you again.

Editing the Description of a Service-Linked Role for Amazon Timestream for InfluxDB

Amazon Timestream for InfluxDB does not allow you to edit the AmazonTimestreamInfluxDBServiceRolePolicy service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM.

Editing a Service-Linked Role Description (IAM Console)

You can use the IAM console to edit a service-linked role description.

To edit the description of a service-linked role (console)

1. In the left navigation pane of the IAM console, choose **Roles**.
2. Choose the name of the role to modify.
3. To the far right of **Role description**, choose **Edit**.
4. Enter a new description in the box and choose **Save**.

Editing a Service-Linked Role Description (IAM CLI)

You can use IAM operations from the AWS Command Line Interface to edit a service-linked role description.

To change the description of a service-linked role (CLI)

1. (Optional) To view the current description for a role, use the AWS CLI for IAM operation `get-role`.

   Example

   ```
   $ aws iam get-role --role-name AmazonTimestreamInfluxDBServiceRolePolicy
   ```

   Use the role name, not the ARN, to refer to roles with the CLI operations. For example, if a role has the following ARN: `arn:aws:iam::123456789012:role/myrole`, refer to the role as **`myrole`**.

2. To update a service-linked role's description, use the AWS CLI for IAM operation `update-role-description`.

   Linux and MacOS

   ```
   $ aws iam update-role-description \
       --role-name AmazonTimestreamInfluxDBServiceRolePolicy \
       --description "new description"
   ```

   Windows

   ```
   $ aws iam update-role-description ^
       --role-name AmazonTimestreamInfluxDBServiceRolePolicy ^
       --description "new description"
   ```

Editing a Service-Linked Role Description (IAM API)

You can use the IAM API to edit a service-linked role description.

To change the description of a service-linked role (API)

1. (Optional) To view the current description for a role, use the IAM API operation GetRole.

   Example

   ```
   https://iam.amazonaws.com/
      ?Action=GetRole
   ```

```
   &RoleName=AmazonTimestreamInfluxDBServiceRolePolicy
   &Version=2010-05-08
   &AUTHPARAMS
```

2. To update a role's description, use the IAM API operation UpdateRoleDescription.

   Example

```
https://iam.amazonaws.com/
   ?Action=UpdateRoleDescription
   &RoleName=AmazonTimestreamInfluxDBServiceRolePolicy
   &Version=2010-05-08
   &Description="New description"
```

Deleting a Service-Linked Role for Amazon Timestream for InfluxDB

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must clean up your service-linked role before you can delete it.

Amazon Timestream for InfluxDB does not delete the service-linked role for you.

Cleaning Up a Service-Linked Role

Before you can use IAM to delete a service-linked role, first confirm that the role has no resources (clusters) associated with it.

To check whether the service-linked role has an active session in the IAM console

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the left navigation pane of the IAM console, choose **Roles**. Then choose the name (not the check box) of the AmazonTimestreamInfluxDBServiceRolePolicy role.
3. On the **Summary** page for the selected role, choose the **Access Advisor** tab.
4. On the **Access Advisor** tab, review recent activity for the service-linked role.

Deleting a Service-Linked Role (IAM Console)

You can use the IAM console to delete a service-linked role.

To delete a service-linked role (console)

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the left navigation pane of the IAM console, choose **Roles**. Then select the check box next to the role name that you want to delete, not the name or row itself.
3. For **Role actions** at the top of the page, choose **Delete role**.
4. In the confirmation page, review the service last accessed data, which shows when each of the selected roles last accessed an AWS service. This helps you to confirm whether the role is currently active. If you want to proceed, choose **Yes, Delete** to submit the service-linked role for deletion.
5. Watch the IAM console notifications to monitor the progress of the service-linked role deletion. Because the IAM service-linked role deletion is asynchronous, after you submit the role for deletion, the deletion task can succeed or fail. If the task fails, you can choose **View details** or **View Resources** from the notifications to learn why the deletion failed.

Deleting a Service-Linked Role (IAM CLI)

You can use IAM operations from the AWS Command Line Interface to delete a service-linked role.

To delete a service-linked role (CLI)

1. If you don't know the name of the service-linked role that you want to delete, enter the following command. This command lists the roles and their Amazon Resource Names (ARNs) in your account.

   ```
   $ aws iam get-role --role-name role-name
   ```

   Use the role name, not the ARN, to refer to roles with the CLI operations. For example, if a role has the ARN `arn:aws:iam::123456789012:role/myrole`, you refer to the role as **`myrole`**.
2. Because a service-linked role cannot be deleted if it is being used or has associated resources, you must submit a deletion request with the delete-service-linked-role command. That request can be denied if these conditions are not met. You must capture the `deletion-task-id` from the response to check the status of the deletion task. Enter the following to submit a service-linked role deletion request.

```
$ aws iam delete-service-linked-role --role-name role-name
```

3. Run the get-service-linked-role-deletion-status command to check the status of the deletion task.

```
$ aws iam get-service-linked-role-deletion-status --deletion-task-id deletion-task-id
```

The status of the deletion task can be `NOT_STARTED`, `IN_PROGRESS`, `SUCCEEDED`, or `FAILED`. If the deletion fails, the call returns the reason that it failed so that you can troubleshoot.

Deleting a Service-Linked Role (IAM API)

You can use the IAM API to delete a service-linked role.

To delete a service-linked role (API)

1. To submit a deletion request for a service-linked roll, call DeleteServiceLinkedRole. In the request, specify a role name.

   Because a service-linked role cannot be deleted if it is being used or has associated resources, you must submit a deletion request. That request can be denied if these conditions are not met. You must capture the `DeletionTaskId` from the response to check the status of the deletion task.

2. To check the status of the deletion, call GetServiceLinkedRoleDeletionStatus. In the request, specify the `DeletionTaskId`.

   The status of the deletion task can be `NOT_STARTED`, `IN_PROGRESS`, `SUCCEEDED`, or `FAILED`. If the deletion fails, the call returns the reason that it failed so that you can troubleshoot.

Supported Regions for Amazon Timestream for InfluxDB Service-Linked Roles

Amazon Timestream for InfluxDB supports using service-linked roles in all of the Regions where the service is available. For more information, see AWS service endpoints.

AWS managed policies for Amazon Timestream for InfluxDB

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the **ReadOnlyAccess** AWS managed policy provides read-only access to all AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the *IAM User Guide*.

AWS managed policy: AmazonTimestreamInfluxDBServiceRolePolicy

You cannot attach the AmazonTimestreamInfluxDBServiceRolePolicy AWS managed policy to identities in your account. This policy is part of the AWS TimestreamforInfluxDB service-linked role. This role allows the service to manage network interfaces and security groups in your account.

Timestream for InfluxDB uses the permissions in this policy to manage EC2 security groups and network interfaces. This is required to manage Timestream for InfluxDB DB instances.

To review this policy in JSON format, see AmazonTimestreamInfluxDBServiceRolePolicy.

AWS-managed policies for Amazon Timestream for InfluxDB

AWS addresses many common use cases by providing standalone IAM policies that are created and administered by AWS. Managed policies grant necessary permissions for common use cases so you can avoid having to investigate what permissions are needed. For more information, see AWS Managed Policies in the *IAM User Guide*.

The following AWS managed policies, which you can attach to users in your account, are specific to Timestream for InfluxDB:

AmazonTimestreamInfluxDBFullAccess

You can attach the `AmazonTimestreamInfluxDBFullAccess` policy to your IAM identities. This policy grants administrative permissions that allow full access to all Timestream for InfluxDB resources.

You can also create your own custom IAM policies to allow permissions for Amazon Timestream for InfluxDB API actions. You can attach these custom policies to the IAM users or groups that require those permissions.

To review this policy in JSON format, see AmazonTimestreamInfluxDBFullAccess.

AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess

You can attach the `AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` policy to your IAM identities. This policy grants administrative permissions that allow full access to all Timestream for InfluxDB resources, excluding any marketplace-related actions.

You can also create your own custom IAM policies to allow permissions for Timestream for InfluxDB API actions. You can attach these custom policies to the IAM users or groups that require those permissions.

To review this policy in JSON format, see AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess.

Timestream for InfluxDB updates to AWS managed policies

View details about updates to AWS managed policies for Timestream for InfluxDB since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Timestream for InfluxDB Document history page.

Change	Description	Date
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `CreateDbBackup`, `GetDbBackup`, `ListDbBackups`, `DeleteDbBackup`, and `RestoreFromDbBackup` actions to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy for backup and restore operations on Amazon Timestream InfluxDB resources.	August 4, 2026
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `CreateDbBackup`, `GetDbBackup`, `ListDbBackups`, `DeleteDbBackup`, and `RestoreFromDbBackup` actions to the existing `AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` managed policy for backup and restore operations on Amazon Timestream InfluxDB resources.	August 4, 2026
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the RebootDbInstance and RebootDbCluster actions to the existing `AmazonTimestreamInfluxDBFul`	12/17/2025

Change	Description	Date
	`lAccess` managed policy for rebooting Amazon Timestream InfluxDB resources.	
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the RebootDbInstance and RebootDbCluster actions to the existing `AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` managed policy for rebooting Amazon Timestream InfluxDB resources.	12/17/2025
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `ec2:DescribeVpcEndpoints` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy for describing the VPC endpoints.	11/13/2025

Change	Description	Date
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `ec2:DescribeVpcEndpoints` action to the existing `AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` managed policy for describing the VPC endpoints.	11/13/2025
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB updated the existing managed policy `AmazonTimestreamInfluxDBFullAccess` that adds necessary permissions to access Marketplace APIs for managing subscription required for creating and updating Timestream for InfluxDB cluster resources.	4/16/2025
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB updated the existing managed policy `AmazonTimestreamInfluxDBFullAccess` that adds marketplace product ID to support subscription to InfluxDB enterprise marketplace offerings for Timestream for InfluxDB cluster resources.	10/17/2025

Change	Description	Date
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB updated the existing managed policy `AmazonTimestreamInfluxDBFullAccess` that adds necessary permissions to access Marketplace APIs for managing subscription required for creating and updating Timestream for InfluxDB cluster resources.	4/16/2025
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – New policy	Amazon Timestream for InfluxDB added a new policy to provide administrative access to manage Amazon Timestream for InfluxDB instances and parameter groups except marketplace operations.	04/16/2025
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB updated the existing managed policy `AmazonTimestreamInfluxDBFullAccess` to also provide full administrative access to create, update, delete, and list Amazon Timestream InfluxDB clusters.	2/17/2025

Change	Description	Date
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Added the `ec2:DescribeRouteTables` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy. This action is used for describing your route tables	10/08/2024
AWS managed policy: AmazonTimestreamInfluxDBServiceRolePolicy – New policy	Amazon Timestream for InfluxDB added a new policy that allows the service to manage network interfaces and security groups in your account.	03/14/2024
AmazonTimestreamInfluxDBFullAccess – New policy	Amazon Timestream for InfluxDB added a new policy to provide full administrative access to create, update, delete and list Amazon Timestream InfluxDB instances and create and list parameter groups.	03/14/2024

Connecting to Timestream for InfluxDB through a VPC endpoint

You can connect directly to Timestream for InfluxDB through a private interface endpoint in your virtual private cloud (VPC). When you use an interface VPC endpoint, communication between your VPC and Timestream for InfluxDB is conducted entirely within the AWS network.

Timestream for InfluxDB supports Amazon Virtual Private Cloud (Amazon VPC) endpoints powered by AWS PrivateLink. Each VPC endpoint is represented by one or more Elastic Network Interfaces (ENIs) with private IP addresses in your VPC subnets.

The interface VPC endpoint connects your VPC directly to Timestream for InfluxDB without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. The instances in your VPC do not need public IP addresses to communicate with Timestream for InfluxDB.

Regions

Timestream for InfluxDB supports VPC endpoints and VPC endpoint policies in all AWS Regions in which Timestream for InfluxDB is supported.

Topics

- Considerations for Timestream for InfluxDB VPC endpoints
- Creating a VPC endpoint for Timestream for InfluxDB
- Connecting to an Timestream for InfluxDB VPC endpoint
- Controlling access to a VPC endpoint
- Using a VPC endpoint in a policy statement
- Logging your VPC endpoint

Considerations for Timestream for InfluxDB VPC endpoints

Before you set up an interface VPC endpoint for Timestream for InfluxDB, review the Interface endpoint properties and limitations topic in the *AWS PrivateLink Guide*.

Timestream for InfluxDB support for a VPC endpoint includes the following.

- You can use your VPC endpoint to call all Timestream for InfluxDB API operations from your VPC.
- You can use AWS CloudTrail logs to audit your use of Timestream for InfluxDB resources through the VPC endpoint. For details, see Logging your VPC endpoint.

Creating a VPC endpoint for Timestream for InfluxDB

You can create a VPC endpoint for Timestream for InfluxDB by using the Amazon VPC console or the Amazon VPC API. For more information, see Create an interface endpoint in the *AWS PrivateLink Guide*.

- To create a VPC endpoint for Timestream for InfluxDB, use the following service name:

```
com.amazonaws.region.timestream-influxdb
```

For example, in the US West (Oregon) Region (`us-west-2`), the service name would be:

```
com.amazonaws.us-west-2.timestream-influxdb
```

To make it easier to use the VPC endpoint, you can enable a private DNS name for your VPC endpoint. If you select the **Enable DNS Name** option, the standard Timestream for InfluxDB DNS hostname resolves to your VPC endpoint. For example, `https://timestream-influxdb.us-west-2.amazonaws.com` would resolve to a VPC endpoint connected to service name `com.amazonaws.us-west-2.timestream-influxdb`.

This option makes it easier to use the VPC endpoint. The AWS SDKs and AWS CLI use the standard Timestream for InfluxDB DNS hostname by default, so you do not need to specify the VPC endpoint URL in applications and commands.

For more information, see Accessing a service through an interface endpoint in the *AWS PrivateLink Guide*.

Connecting to an Timestream for InfluxDB VPC endpoint

You can connect to Timestream for InfluxDB through the VPC endpoint by using an AWS SDK, the AWS CLI or AWS Tools for PowerShell. To specify the VPC endpoint, use its DNS name.

If you enabled private hostnames when you created your VPC endpoint, you do not need to specify the VPC endpoint URL in your CLI commands or application configuration. The standard Timestream for InfluxDB DNS hostname resolves to your VPC endpoint. The AWS CLI and SDKs use this hostname by default, so you can begin using the VPC endpoint to connect to an Timestream for InfluxDB regional endpoint without changing anything in your scripts and applications.

To use private hostnames, the `enableDnsHostnames` and `enableDnsSupport` attributes of your VPC must be set to `true`. To set these attributes, use the ModifyVpcAttribute operation. For details, see View and update DNS attributes for your VPC in the *Amazon VPC User Guide*.

Controlling access to a VPC endpoint

To control access to your VPC endpoint for Timestream for InfluxDB, attach a *VPC endpoint policy* to your VPC endpoint. The endpoint policy determines whether principals can use the VPC endpoint to call Timestream for InfluxDB operations on Timestream for InfluxDB resources.

You can create a VPC endpoint policy when you create your endpoint, and you can change the VPC endpoint policy at any time. Use the VPC management console, or the CreateVpcEndpoint or ModifyVpcEndpoint operations. You can also create and change a VPC endpoint policy by using an AWS CloudFormation template. For help using the VPC management console, see Create an interface endpoint and Modifying an interface endpoint in the *AWS PrivateLink Guide*.

> **Note**
>
> Timestream for InfluxDB supports VPC endpoint policies beginning in July 2020. VPC endpoints for Timestream for InfluxDB that were created before that date have the default VPC endpoint policy, but you can change it at any time.

Topics

- About VPC endpoint policies
- Default VPC endpoint policy
- Creating a VPC endpoint policy
- Viewing a VPC endpoint policy

About VPC endpoint policies

For an Timestream for InfluxDB request that uses a VPC endpoint to be successful, the principal requires permissions from two sources:

- A IAM policy must give principal permission to call the operation on the resource.
- A VPC endpoint policy must give the principal permission to use the endpoint to make the request.

Default VPC endpoint policy

Every VPC endpoint has a VPC endpoint policy, but you are not required to specify the policy. If you don't specify a policy, the default endpoint policy allows all operations by all principals on all resources over the endpoint.

However, for Timestream for InfluxDB resources, the principal must also have permission to call the operation from an IAM policy Therefore, in practice, the default policy says that if a principal has permission to call an operation on a resource, they can also call it by using the endpoint.

```
{
  "Statement": [
    {
      "Action": "*",
      "Effect": "Allow",
      "Principal": "*",
      "Resource": "*"
    }
  ]
}
```

To allow principals to use the VPC endpoint for only a subset of their permitted operations, create or update the VPC endpoint policy.

Creating a VPC endpoint policy

A VPC endpoint policy determines whether a principal has permission to use the VPC endpoint to perform operations on a resource. For Timestream for InfluxDB resources, the principal must also have permission to perform the operations from a IAM policy,

Each VPC endpoint policy statement requires the following elements:

- The principal that can perform actions
- The actions that can be performed
- The resources on which actions can be performed

The policy statement doesn't specify the VPC endpoint. Instead, it applies to any VPC endpoint to which the policy is attached. For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

AWS CloudTrail logs all operations that use the VPC endpoint.

Viewing a VPC endpoint policy

To view the VPC endpoint policy for an endpoint, use the VPC management console or the DescribeVpcEndpoints operation.

The following AWS CLI command gets the policy for the endpoint with the specified VPC endpoint ID.

Before using this command, replace the example endpoint ID with a valid one from your account.

```
$ aws ec2 describe-vpc-endpoints \

--query 'VpcEndpoints[?VpcEndpointId==`vpc-endpoint-id`].[PolicyDocument]'

--output text
```

Using a VPC endpoint in a policy statement

You can control access to Timestream for InfluxDB resources and operations when the request comes from VPC or uses a VPC endpoint. To do so, use one of the following global condition keys in a IAM policy.

- Use the `aws:sourceVpce` condition key to grant or restrict access based on the VPC endpoint.
- Use the `aws:sourceVpc` condition key to grant or restrict access based on the VPC that hosts the private endpoint.

> **Note**
>
> Use caution when creating key policies and IAM policies based on your VPC endpoint. If a policy statement requires that requests come from a particular VPC or VPC endpoint, requests from integrated AWS services that use an Timestream for InfluxDB resource on your behalf might fail.
> Also, the `aws:sourceIP` condition key is not effective when the request comes from an Amazon VPC endpoint. To restrict requests to a VPC endpoint, use the `aws:sourceVpce` or `aws:sourceVpc` condition keys. For more information, see Identity and access management for VPC endpoints and VPC endpoint services in the *AWS PrivateLink Guide*.

You can use these global condition keys to control access to operations like CreateDbInstance that don't depend on any particular resource.

Logging your VPC endpoint

AWS CloudTrail logs all operations that use the VPC endpoint. When a request to Timestream for InfluxDB uses a VPC endpoint, the VPC endpoint ID appears in the AWS CloudTrail log entry that records the request. You can use the endpoint ID to audit the use of your Timestream for InfluxDB VPC endpoint.

However, your CloudTrail logs don't include operations requested by principals in other accounts or requests for Timestream for InfluxDB operations on Timestream for InfluxDB resources and aliases in other accounts. Also, to protect your VPC, requests that are denied by a VPC endpoint policy, but otherwise would have been allowed, are not recorded in AWS CloudTrail.

Logging and monitoring in Timestream for InfluxDB

Monitoring is an important part of maintaining the reliability, availability, and performance of Timestream for InfluxDB and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. However, before you start monitoring Timestream for InfluxDB, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal Timestream for InfluxDB performance in your environment, by measuring performance at various times and under different load conditions. As you monitor Timestream for InfluxDB, store historical monitoring data so that you can compare it with current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish a baseline, you should, at a minimum, monitor the following items:

- System errors, so that you can determine whether any requests resulted in an error.

Topics

- Monitoring tools
- Logging Timestream for InfluxDB API calls with AWS CloudTrail

Monitoring tools

AWS provides various tools that you can use to monitor Timestream for InfluxDB. You can configure some of these tools to do the monitoring for you, while some of the tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Topics

- Automated monitoring tools
- Manual monitoring tools

Automated monitoring tools

You can use the following automated monitoring tools to watch Timestream for InfluxDB and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see Monitoring with Amazon CloudWatch.

Manual monitoring tools

Another important part of monitoring Timestream for InfluxDB involves manually monitoring those items that the CloudWatch alarms don't cover. The Timestream for InfluxDB, CloudWatch, Trusted Advisor, and other AWS Management Console dashboards provide an at-a-glance view of the state of your AWS environment.

- The CloudWatch home page shows the following:
  - Current alarms and status
  - Graphs of alarms and resources
  - Service health status

  In addition, you can use CloudWatch to do the following:
  - Create customized dashboards to monitor the services you care about

- Graph metric data to troubleshoot issues and discover trends
- Search and browse all your AWS resource metrics
- Create and edit alarms to be notified of problems

Logging Timestream for InfluxDB API calls with AWS CloudTrail

Timestream for InfluxDB is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Timestream for InfluxDB. CloudTrail captures Data Definition Language (DDL) API calls for Timestream for InfluxDB as events. The calls that are captured include calls from the Timestream for InfluxDB console and code calls to the Timestream for InfluxDB API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for Timestream for InfluxDB. If you don't configure a trail, you can still view the most recent events on the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Timestream for InfluxDB, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Timestream for InfluxDB information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Timestream for InfluxDB, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Timestream for InfluxDB, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.

For more information, see the following topics in the *AWS CloudTrail User Guide*:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations

- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions
- Receiving CloudTrail Log Files from Multiple Accounts
- Logging data events

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials
- Whether the request was made with temporary security credentials for a role or federated user
- Whether the request was made by another AWS service

For more information, see the CloudTrail userIdentity Element.

Compliance validation for Amazon Timestream for InfluxDB

Third-party auditors assess the security and compliance of Amazon Timestream for InfluxDB as part of multiple AWS compliance programs. These include the following:

- GDPR
- HIPAA
- PCI
- SOC

Resilience in Amazon Timestream for InfluxDB

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Amazon Timestream for InfluxDB periodically takes internal backups and retains them for 24 hours to support availability and durability. Snapshots are taken during deletes and retained for 30 days to support restores. To access or use these, file a ticket at AWS support.

You can create your instance with Multi-AZ recovery capabilities. For more information, see Multi-AZ DB instance deployments.

Infrastructure security in Amazon Timestream for InfluxDB

As a managed service, Amazon Timestream for InfluxDB is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published control plane API calls to access Timestream for InfluxDB through the network. For more information, see Control planes and data planes. Clients must support Transport Layer Security (TLS) 1.2 or later. We recommend TLS 1.2 or 1.3. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Timestream for InfluxDB is architected so that your traffic is isolated to the specific AWS Region that your Timestream for InfluxDB instance resides in.

Security groups

Security groups control the access that traffic has in and out of a DB instance. By default, network access is turned off to a DB instance. You can specify rules in a security group that allow access from an IP address range, port, or security group. After ingress rules are configured, the same rules apply to all DB instances that are associated with that security group.

For more information, see Controlling access to a DB instance in a VPC.

Configuration and vulnerability analysis in Timestream for InfluxDB

Configuration and IT controls are a shared responsibility between AWS and you, our customer. For more information, see the AWS shared responsibility model. In addition to the shared responsibility model, Timestream for InfluxDB users should be aware of the following:

- It is the customer responsibility to patch their client applications with the relevant client side dependencies.
- Customers should consider penetration testing if appropriate (see https://aws.amazon.com/security/penetration-testing/.)

Incident response in Timestream for InfluxDB

Amazon Timestream for InfluxDB service incidents are reported in the Personal Health Dashboard. You can learn more about the dashboard and AWS Health here.

Timestream for InfluxDB supports reporting using AWS CloudTrail. For more information, see Logging Timestream for InfluxDB API calls with AWS CloudTrail.

Amazon Timestream for InfluxDB API and interface VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon Amazon Timestream for InfluxDB control plane API endpoints by creating an *interface VPC endpoint*. Interface endpoints are powered by AWS PrivateLink. AWS PrivateLink allows you to privately access Amazon Timestream for InfluxDB API operations without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection.

Instances in your VPC don't need public IP addresses to communicate with Amazon Timestream for InfluxDB API endpoints. Your instances also don't need public IP addresses to use any of the available Timestream for InfluxDB API operations. Traffic between your VPC and Amazon Timestream for InfluxDB doesn't leave the Amazon network. Each interface endpoint is represented by one or more elastic network interfaces in your subnets. For more information on elastic network interfaces, see Elastic network interfaces in the *Amazon EC2 User Guide*.

- For more information about VPC endpoints, see Interface VPC endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*.
- For more information about Timestream for InfluxDB API operations, see Timestream for InfluxDB API operations.

After you create an interface VPC endpoint, if you enable private DNS hostnames for the endpoint, the default Timestream for InfluxDB endpoint (https://timestream-

influxb.*Region*.amazonaws.com) resolves to your VPC endpoint. If you do not enable private DNS hostnames, Amazon VPC provides a DNS endpoint name that you can use in the following format:

```
VPC_Endpoint_ID.timestream-influxb.Region.vpce.amazonaws.com
```

For more information, see Interface VPC Endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*. Timestream for InfluxDB supports making calls to all of its API Actions inside your VPC.

> **Note**
>
> Private DNS hostnames can be enabled for only one VPC endpoint in the VPC. If you want to create an additional VPC endpoint then private DNS hostname should be disabled for it.

Considerations for VPC endpoints

Before you set up an interface VPC endpoint for Amazon Timestream for InfluxDB API endpoints, ensure that you review Interface endpoint properties and limitations in the *Amazon VPC User Guide*. All Timestream for InfluxDB API operations that are relevant to managing Amazon Timestream for InfluxDB resources are available from your VPC using AWS PrivateLink. VPC endpoint policies are supported for Timestream for InfluxDB API endpoints. By default, full access to Timestream for InfluxDB API operations is allowed through the endpoint. For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Creating an interface VPC endpoint for the Timestream for InfluxDB API

You can create a VPC endpoint for the Amazon Timestream for InfluxDB API using either the Amazon VPC console or the AWS CLI. For more information, see Creating an interface endpoint in the *Amazon VPC User Guide*.

After you create an interface VPC endpoint, you can enable private DNS host names for the endpoint. When you do, the default Amazon Timestream for InfluxDB endpoint (https://timestream-influxb.*Region*.amazonaws.com) resolves to your VPC endpoint. For more information, see Accessing a service through an interface endpoint in the *Amazon VPC User Guide*.

Creating a VPC endpoint policy for the Amazon Timestream for InfluxDB API

You can attach an endpoint policy to your VPC endpoint that controls access to the Timestream for InfluxDB API. The policy specifies the following:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Example VPC endpoint policy for Timestream for InfluxDB API actions

The following is an example of an endpoint policy for the Timestream for InfluxDB API. When attached to an endpoint, this policy grants access to the listed Timestream for InfluxDB API actions for all principals on all resources.

```
{
 "Statement": [{
  "Principal": "*",
  "Effect": "Allow",
  "Action": [
   "timestream-influxb:CreateDbInstance",
   "timestream-influxb:UpdateDbInstance"
  ],
  "Resource": "*"
 }]
}
```

Example VPC endpoint policy that denies all access from a specified AWS account

The following VPC endpoint policy denies AWS account *123456789012* all access to resources using the endpoint. The policy allows all actions from other accounts.

```
{
 "Statement": [{
   "Action": "*",
   "Effect": "Allow",
   "Resource": "*",
   "Principal": "*"
  },
  {
   "Action": "*",
   "Effect": "Deny",
   "Resource": "*",
```

```
    "Principal": {
     "AWS": [
      "123456789012"
     ]
    }
   }
  ]
 }
```

Security best practices for Timestream for InfluxDB

Amazon Timestream for InfluxDB provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don’t represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Implement least privilege access

When granting permissions, you decide who is getting what permissions to which Timestream for InfluxDB resources. You enable specific actions that you want to allow on those resources. Therefore you should grant only the permissions that are required to perform a task. Implementing least privilege access is fundamental in reducing security risk and the impact that could result from errors or malicious intent.

Use IAM roles

Producer and client applications must have valid credentials to access Timestream for InfluxDB DB instances. You should not store AWS credentials directly in a client application or in an Amazon S3 bucket. These are long-term credentials that are not automatically rotated and could have a significant business impact if they are compromised.

Instead, you should use an IAM role to manage temporary credentials for your producer and client applications to access Timestream for InfluxDB DB instances. When you use a role, you don't have to use long-term credentials (such as a user name and password or access keys) to access other resources.

For more information, see the following topics in the *IAM User Guide*:

- IAM Roles

- Common Scenarios for Roles: Users, Applications, and Services

Use AWS Identity and Access Management (IAM) accounts to control access to Amazon Timestream for InfluxDB API operations, especially operations that create, modify, or delete Amazon Timestream for InfluxDB resources. Such resources include DB instances, security groups, and parameter groups.

- Create an individual user for each person who manages Amazon Timestream for InfluxDB resources, including yourself. Don't use AWS root credentials to manage Amazon Timestream for InfluxDB resources.
- Grant each user the minimum set of permissions required to perform his or her duties.
- Use IAM groups to effectively manage permissions for multiple users.
- Rotate your IAM credentials regularly.
- Configure AWS Secrets Manager to automatically rotate the secrets for Amazon Timestream for InfluxDB. For more information, see Rotating your AWS Secrets Manager secrets in the *AWS Secrets Manager User Guide*. You can also retrieve the credential from AWS Secrets Manager programmatically. For more information, see Retrieving the secret value in the *AWS Secrets Manager User Guide*.
- Secure your Timestream for InfluxDB influx API tokens by using the API tokens.

Implement Server-Side Encryption in Dependent Resources

Data at rest and data in transit can be encrypted in Timestream for InfluxDB. For more information, see Encryption in transit.

Use CloudTrail to Monitor API Calls

Timestream for InfluxDB is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Timestream for InfluxDB.

Using the information collected by CloudTrail, you can determine the request that was made to Timestream for InfluxDB, the IP address from which the request was made, who made the request, when it was made, and additional details.

For more information, see the section called “Logging Timestream for LiveAnalytics API calls with AWS CloudTrail”.

Amazon Timestream for InfluxDB supports control plane CloudTrail events, but not data plane. For more information, see Control planes and data planes.

Public accessibility

When you launch a DB instance inside a virtual private cloud (VPC) based on the Amazon VPC service, you can turn on or off public accessibility for that DB instance. To designate whether the DB instance that you create has a DNS name that resolves to a public IP address, you use the Public accessibility parameter. By using this parameter, you can designate whether there is public access to the DB instance

If your DB instance is in a VPC but isn't publicly accessible, you can also use an AWS Site-to-Site VPN connection or an AWS Direct Connect connection to access it from a private network.

If your DB instance is publicly accessible, be sure to take steps to prevent or help mitigate denial of service related threats. For more information, see Introduction to denial of service attacks and Protecting networks.

Working with other services

Amazon Timestream for InfluxDB integrates with a variety of AWS services and popular third-party tools. All services and tools compatible with open-source InfluxDB should work seamlessly with Timestream for InfluxDB. Among those we would like to note:

Topics

- DBeaver
- Grafana

DBeaver

DBeaver is a free universal SQL client that can be used to manage any database that has a JDBC driver. It is widely used among developers and database administrators because of its robust data viewing, editing, and management capabilities. Using DBeaver's cloud connectivity options, you can connect DBeaver to Amazon Timestream for InfluxDB natively. DBeaver provides a comprehensive and intuitive interface to work with time series data directly from within a DBeaver application. Using your credentials, it also gives you full access to any queries that you could execute from another query interface. It even lets you create graphs for better understanding and visualization of query results.

To configure your DBeaver client to connect to your Timestream for InfluxDB DB instance or cluster, refer to DBeaver's guide to InfluxDB configuration.

Grafana

Use Amazon Managed Grafana, Grafana, or Grafana Cloud to visualize data from your Timestream for InfluxDB instance.

Connect to Grafana

> **Important**
>
> The instructions in this guide require Grafana Cloud or Grafana 10.3+.

1. Create your Timestream for InfluxDB DB instance or Timestream for InfluxDB DB cluster.
2. Create an Amazon Managed Grafana workspace, sign up for Grafana Cloud, or download and install Grafana.
3. Visit your Amazon Managed Grafana, Grafana Cloud user interface (UI) or, if running Grafana locally, start Grafana and visit http://localhost:3000 in your browser.
4. In the left navigation of the Grafana UI, open the **Connections** section and select **Add new connection**.
5. Select **InfluxDB** from the list of available data sources and click **Add new data source**.
6. On the Data Source configuration page, enter a name for your InfluxDB data source.
7. In the **Query Language** dropdown list, select one of the query languages supported by InfluxDB 2.7 (**Flux** or **InfluxQL**).

> **Important**
>
> SQL is only supported in InfluxDB 3.

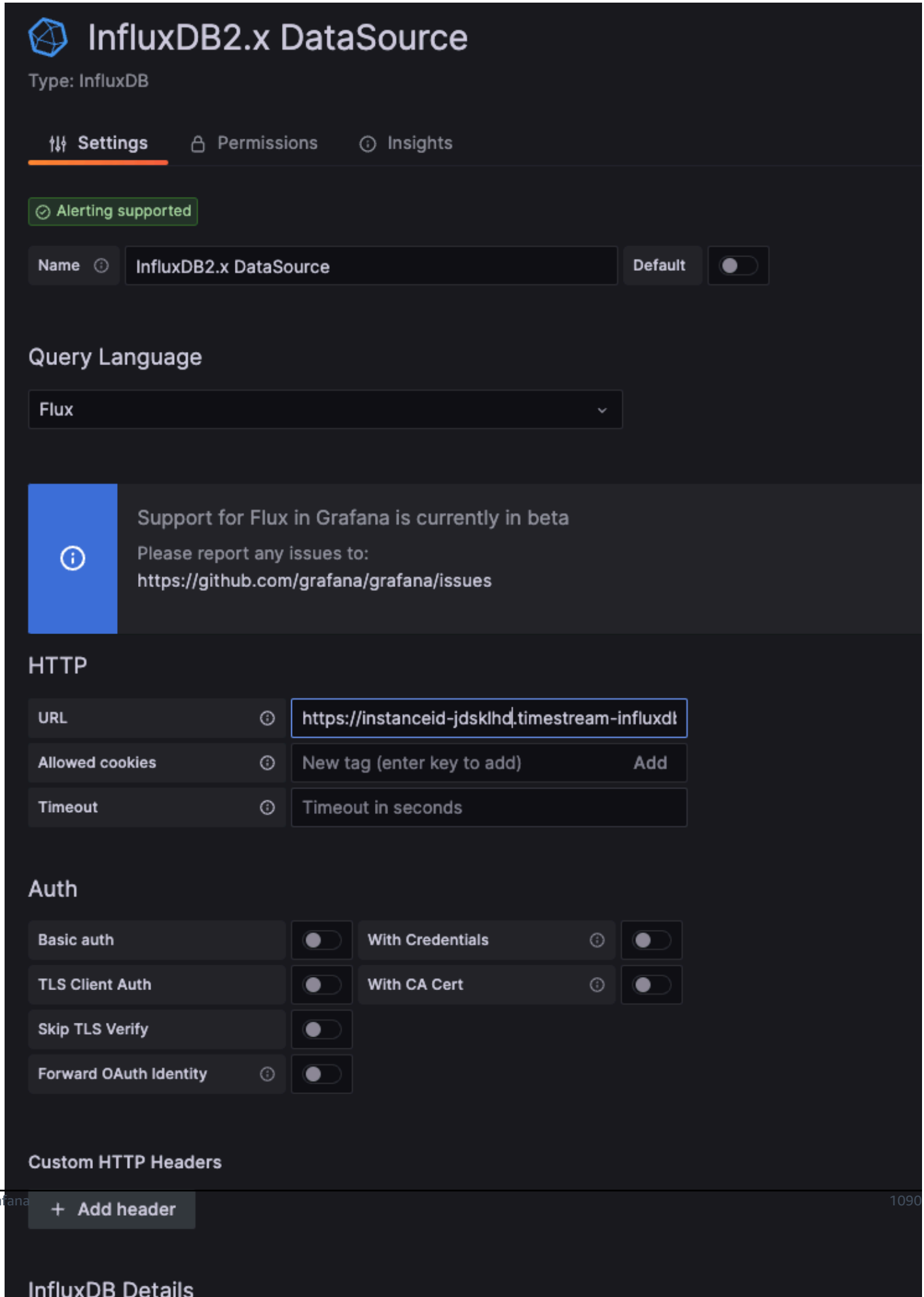
Configure Grafana to use Flux

With Flux selected as the query language in your InfluxDB data source, configure your InfluxDB connection:

1. In the **HTTP** section, enter your InfluxDB URL in the **URL** field.

```
https://your-timestream-for-influxdb-endpoint:8086
```

2. In the **InfluxDB Details** section, enter the following:
   - In **Organization**: Your InfluxDB organization name or ID.
   - In **Token**: Your InfluxDB API token.
   - In **Default Bucket**: The default bucket to use in Flux queries.
   - In **Min time interval**: The Grafana minimum time interval. The default is 10 seconds.
   - In **Max series**: The maximum number of series or tables Grafana will process. The default is 1,000.


InfluxDB2.x DataSource
Type: InfluxDB
Settings
Permissions
Insights
Alerting supported
Name
InfluxDB2.x DataSource
Default
Query Language
Flux
Support for Flux in Grafana is currently in beta
Please report any issues to:
https://github.com/grafana/grafana/issues
HTTP
URL
https://instanceid-jdsklhd.timestream-influxdb
Allowed cookies
New tag (enter key to add)
Add
Timeout
Timeout in seconds
Auth
Basic auth
With Credentials
TLS Client Auth
With CA Cert
Skip TLS Verify
Forward OAuth Identity
Custom HTTP Headers
+ Add header
InfluxDB Details

3. Click **Save & test**. Grafana attempts to connect to the InfluxDB 2.7 data source and returns the results of the test.

Configure Grafana to use InfluxQL

To query InfluxDB 2.7 with InfluxQL, find your use case below and then complete the instructions to configure Grafana.

New install of InfluxDB 2.7:

To configure Grafana to use InfluxQL with a new install of InfluxDB 2.7, do the following:

1. Authenticate with InfluxDB 2.7 tokens.
2. Manually create DBRP mappings.

Manual migration from InfluxDB 1.x to 2.7:

To configure Grafana to use InfluxQL when you have manually migrated from InfluxDB 1.x to InfluxDB 2.7, do the following:

1. If your InfluxDB 1.x instance required authentication, create v1-compatible authentication credentials to match your previous 1.x username and password. Otherwise, use InfluxDB v2 token authentication.
2. Manually create DBRP mappings.

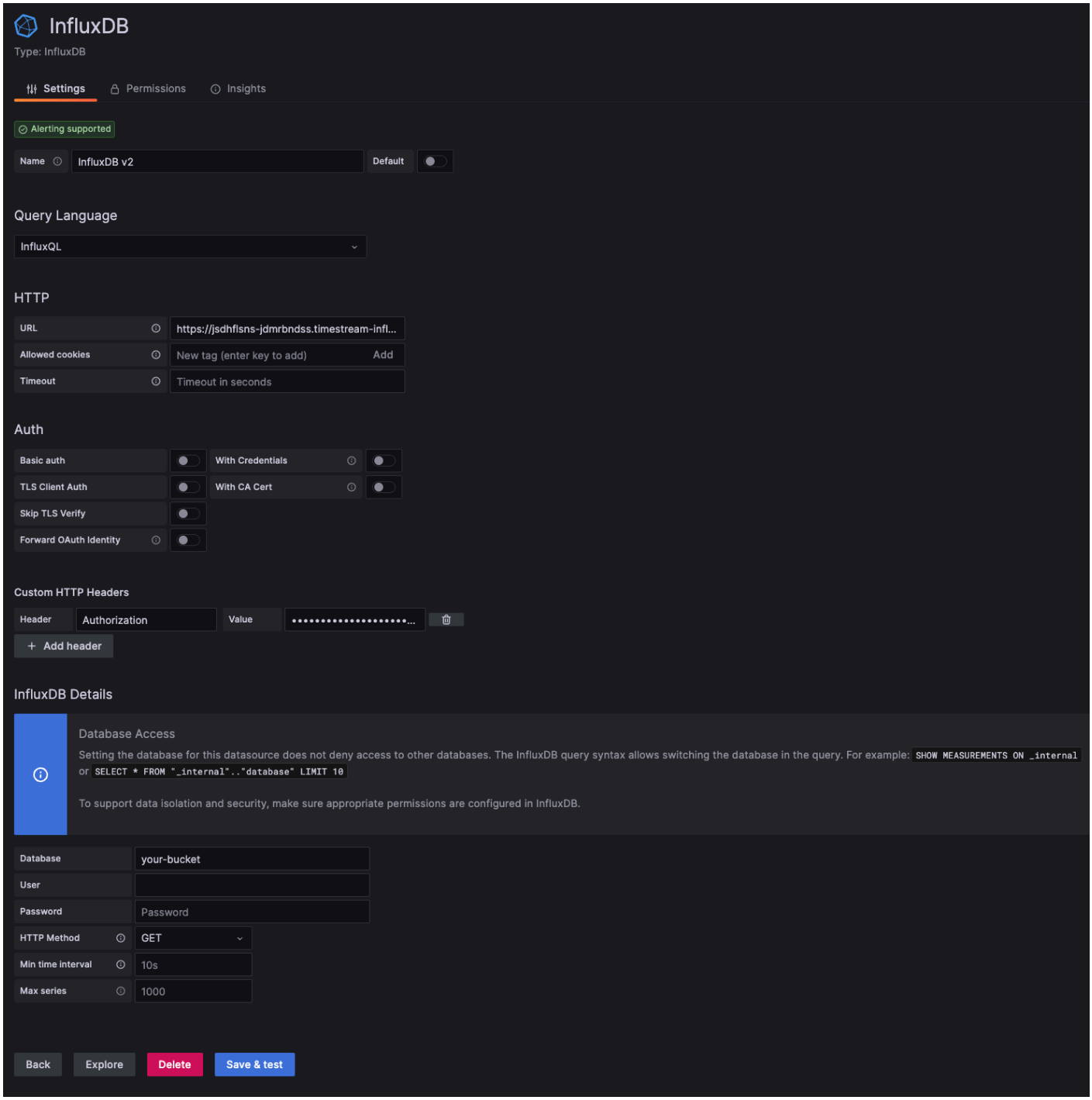
With InfluxQL selected as the query language in your InfluxDB data source, configure your InfluxDB connection:

1. In the **HTTP** section, enter your InfluxDB URL in the **URL** field.

```
https://your-timestream-for-influxdb-endpoint:8086
```

2. In the **Custom HTTP Headers** section, enter the following:
   - Select **Add header**. Provide your InfluxDB API token:
     - In **Header**, enter **Authorization**.
     - In **Value**, use the `Token` schema and provide your InfluxDB API token. For example, `Token y0uR5uP3rSecr3tT0k3n`.
3. In the **InfluxDB Details** section, enter the following:

- In **Database**: The database name mapped to your InfluxDB 2.7 bucket.
- In **User** and **Password**: The username and password associated with your InfluxDB 1.x compatibility authorization.
- In **HTTP Method**: Select **GET**.



4. Click **Save & test**. Grafana attempts to connect to the InfluxDB 2.7 data source and returns the results of the test.

Query and visualize data

After configuring your InfluxDB connection, you can use Grafana and Flux to query and visualize time series data stored in your InfluxDB instance.

For more information about using Grafana, see the Grafana technical documentation. If you are just learning Flux, see Get started with Flux.

API reference

For a complete list and details of Amazon Timestream for InfluxDB APIs, see Amazon Timestream for InfluxDB APIs.

For error codes common to all AWS services, see the AWS Support section.

Document history

Change	Description	Date
`AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added the `CreateDbBackup`, `GetDbBackup`, `ListDbBackups`, `DeleteDbBackup`, and `RestoreFromDbBackup` actions to the `AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` managed policy. These actions support backup and restore operations on Amazon Timestream for InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	August 4, 2026

AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `CreateDbBackup`, `GetDbBackup`, `ListDbBackups`, `DeleteDbBackup`, and `RestoreFromDbBackup` actions to the `AmazonTimestreamInfluxDBFullAccess` managed policy. These actions support backup and restore operations on Amazon Timestream for InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	August 4, 2026
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `RebootDbInstance` and `RebootDbCluster` actions to the existing `AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` managed policy for rebooting Amazon Timestream InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	December 17, 2025

AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the RebootDbInstance and RebootDbCluster actions to the existing AmazonTimestreamInfluxDBFullAccess managed policy for rebooting Amazon Timestream InfluxDB resources. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	December 17, 2025
AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the ec2:DescribeVpcEndpoints action to the existing AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess managed policy for describing the VPC endpoints. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	November 13, 2025

AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the `ec2:DescribeVpcEndpoints` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy for describing the VPC endpoints. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	November 13, 2025
AmazonTimestreamInfluxDBFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added Influx Enterprise marketplace product ID to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy to support subscription to enterprise marketplace offerings. See AmazonTimestreamInfluxDBFullAccess.	October 17, 2025
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Timestream for LiveAnalytics has added the AWS Marketplace permissions to the `AmazonTimestreamConsoleFullAccess` managed policy to access marketplace resources and create agreements for InfluxDB Cluster with Read Replicas creation.	August 20, 2025

AmazonTimestreamConsoleFullAccess – Update to an existing policy	Timestream for InfluxDB has added the AWS Marketplace permissions to the AmazonTimestreamConsoleFullAccess managed policy to access marketplace resources and create agreements for InfluxDB Cluster with Read Replicas creation.	August 20, 2025
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Amazon Timestream for InfluxDB has added the pricing:GetProducts permission to the AmazonTimestreamConsoleFullAccess managed policy to provide pricing estimations for InfluxDB resource configurations during creation.	June 10, 2025

Amazon Timestream for LiveAnalytics will no longer be open to new customers starting June 20, 2025.	For similar capabilities to Amazon Timestream for LiveAnalytics, consider Amazon Timestream for InfluxDB 3 Enterprise for large-scale time-series analytics with unlimited cardinality, native SQL support, and multi-nod e scaling. For operational workloads requiring single-digit millisecond query response times, consider Amazon Timestream for InfluxDB 2. Learn more here.	May 20, 2025
`AmazonTimestreamInfluxDBFullAccessWithoutMarketplaceAccess` – New policy	This policy grants administr ative permissions that allow full access to all Timestrea m for InfluxDB resources , excluding any marketpla ce-related actions. For more information see AWS managed policies for Amazon Timestream for InfluxDB.	April 16, 2025
`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	April 16, 2025

`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added access to create, update, delete, and list Amazon Timestream InfluxDB clusters to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	February 17, 2025
Documentation-only update	Updated the Quotas topic to segregate the default quotas and system limits.	October 22, 2024
Amazon Timestream now supports query insights	Timestream now includes support for the query insights feature that helps you optimize your queries, improve their performance, and reduce costs.	October 22, 2024
Amazon Timestream for InfluxDB update to an existing policy.	Amazon Timestream for InfluxDB has added the `ec2:DescribeRouteTables` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy for describing your route tables. For more information, see AWS managed policies for Amazon Timestream for InfluxDB.	October 8, 2024

`AmazonTimestreamInfluxDBFullAccess` – Update to an existing policy	Amazon Timestream for InfluxDB has added the `ec2:DescribeRouteTables` action to the existing `AmazonTimestreamInfluxDBFullAccess` managed policy. This action is used for describing your route tables. See AmazonTimestreamInfluxDBFullAccess.	September 12, 2024
`AmazonTimestreamReadOnlyAccess` – Update to an existing policy	Timestream for LiveAnalytics has added the `DescribeAccountSettings` permission to the `AmazonTimestreamReadOnlyAccess` managed policy for describing AWS account settings.	June 3, 2024
Amazon Timestream for LiveAnalytics now supports Timestream Compute Units (TCUs)	Amazon Timestream for LiveAnalytics now includes support for Timestream Compute Units (TCUs) to measure the compute capacity allocated for your query needs.	April 29, 2024

New policies added	Amazon Timestream for InfluxDB added two new policies: One that allows the service to manage network interfaces and security groups in your account. For more information, see AmazonTimestreamInfluxDBServiceRolePolicy. Another that provide full administrative access to create, update, delete and list Amazon Timestream InfluxDB instances and create and list parameter groups. For more information, see AmazonTimestreamInfluxDBFullAccess.	March 14, 2024
Amazon Timestream for InfluxDB is now generally available.	This documentation covers the initial release of Amazon Timestream for InfluxDB.	March 14, 2024
Amazon Timestream for LiveAnalytics Query events are available in AWS CloudTrail	Amazon Timestream for LiveAnalytics now publishes Query API data events to AWS CloudTrail. Customers can audit all Query API requests made in their AWS accounts, and see information such as which IAM User/Role made the request, when the request was made, which databases and tables were queried, and the request's Query ID.	September 12, 2023

Amazon Timestream for LiveAnalytics UNLOAD	Amazon Timestream for LiveAnalytics now supports UNLOAD to export query results to S3.	May 12, 2023
Amazon Timestream for LiveAnalytics update to an existing policy.	Batch load permissions added to a managed policy.	February 24, 2023
Amazon Timestream for LiveAnalytics batch load.	Amazon Timestream for LiveAnalytics now supports batch load functionality.	February 24, 2023
Amazon Timestream for LiveAnalytics now supports AWS Backup.	Amazon Timestream for LiveAnalytics now supports AWS Backup.	December 14, 2022
Amazon Timestream for LiveAnalytics updates to AWS managed policies	New information about AWS managed policies and Amazon Timestream for LiveAnalytics, including updates to existing managed policies.	November 29, 2021
Amazon Timestream for LiveAnalytics supports scheduled queries	Amazon Timestream for LiveAnalytics now supports running a query on your behalf, based on a schedule.	November 29, 2021
Amazon Timestream for LiveAnalytics supports magnetic store.	Amazon Timestream for LiveAnalytics now supports using magnetic storage for your table writes.	November 29, 2021
Amazon Timestream for LiveAnalytics multi-measure records.	Amazon Timestream for LiveAnalytics now supports a more compact format for storing your time-series data.	November 29, 2021

Amazon Timestream for LiveAnalytics updates to AWS managed policies	New information about AWS managed policies and Amazon Timestream for LiveAnalytics, including updates to existing managed policies.	May 24, 2021
Amazon Timestream for LiveAnalytics is now available in the Europe (Frankfurt) region.	Amazon Timestream for LiveAnalytics is now generally available in the Europe (Frankfurt) region (`eu-central-1`).	April 23, 2021
Amazon Timestream for LiveAnalytics now supports VPC endpoints (AWS PrivateLink).	Amazon Timestream for LiveAnalytics now supports the use of VPC endpoints (AWS PrivateLink).	March 23, 2021
Amazon Timestream now supports cross table queries.	You can use Amazon Timestream for LiveAnalytics to run cross table queries.	February 10, 2021
Amazon Timestream for LiveAnalytics now supports enhanced query execution statistics.	Amazon Timestream for LiveAnalytics now supports enhanced query execution statistics, such as amount of data scanned.	February 10, 2021
Amazon Timestream for LiveAnalytics now supports advanced time series functions.	You can use Amazon Timestream for LiveAnalytics to run SQL queries with advanced time series functions, such as derivatives, integrals, and correlations.	February 10, 2021

Amazon Timestream for LiveAnalytics is now HIPAA, ISO, and PCI compliant.	You can now use Amazon Timestream for LiveAnalytics for workloads that require HIPAA, ISO, and PCI-compliant infrastructure.	January 27, 2021
Amazon Timestream for LiveAnalytics now supports open-source Telegraf and Grafana.	You can now use Telegraf, the open-source, plugin-driven server agent for collecting and reporting metrics, and Grafana, the open-source analytics and monitoring platform for databases, with Amazon Timestream for LiveAnalytics.	November 25, 2020
Amazon Timestream for LiveAnalytics is now generally available.	This documentation covers the initial release of Amazon Timestream for LiveAnalytics.	September 30, 2020

Amazon Timestream for InfluxDB 3

What is Timestream for InfluxDB 3?

Amazon Timestream for InfluxDB 3 is a managed time-series database service that makes it easy for application developers and DevOps teams to run InfluxDB 3 databases on AWS for large-scale time-series analytics applications using open-source APIs. With Amazon Timestream for InfluxDB 3, you can easily set up, operate, and scale time-series workloads designed to handle high-cardinality data and complex analytical queries.

InfluxDB 3 represents a complete architectural reimagining of the InfluxDB database engine. Unlike versions 1 and 2, which utilized a Time-Structured Merge tree (TSM) storage engine, InfluxDB 3 is built from the ground up on entirely different technology foundations. This new version leverages Apache Arrow for in-memory data processing, Apache Data Fusion for query execution, and a columnar storage format (Parquet) for data persistence in object storage (Amazon S3). This architectural shift enables InfluxDB 3 to deliver enhanced performance for high-cardinality data and scale efficiently for large analytical workloads. InfluxDB 3 architecture provides improved query performance and resource utilization for data-intensive use cases.

Amazon Timestream for InfluxDB 3 gives you access to the capabilities of this next-generation time-series database engine. The architectural transformation enables large scale time-series analytics workloads and capitalizes on the compression, partitioning, and query optimization capabilities inherent to the columnar Parquet storage format. By decoupling compute from storage, InfluxDB 3 can scale to handle virtually unlimited data volumes while maintaining cost efficiency.

Amazon Timestream for InfluxDB 3 can automatically back up your database and keep your database software up to date with the latest version. As with all AWS services, there are no upfront investments required, and you pay only for the resources you use.

DB Clusters

A DB cluster is the fundamental building block of Timestream for InfluxDB 3. Unlike traditional database instances, InfluxDB 3 uses a cluster-based architecture that separates compute from storage, leveraging Amazon S3 for virtually unlimited, cost-effective data storage.

You can set up your InfluxDB 3 deployments in either Core or Enterprise versions:

Core Version

The Core version of InfluxDB 3 is limited to single-node cluster deployments and lacks compaction capabilities, which impacts its suitability for certain workloads. Due to these limitations, the Core version is primarily designed for near real-time workloads focused on recent data (typically a few days old). It is not recommended for use cases involving long-term storage and analysis, as performance will degrade over time without compaction processes to optimize storage.

Enterprise Version

The Enterprise version of InfluxDB 3 supports multi-node cluster configurations and includes essential compaction capabilities, enabling horizontal scaling for both read and write operations. These multi-node deployments provide enhanced availability, improved performance for concurrent queries, and greater overall system resilience. The compaction functionality in the Enterprise version makes it suitable for high-cardinality data and long-term analytics use cases, as it continuously optimizes the underlying storage format. Enterprise clusters can be scaled by adding nodes to accommodate growing workloads without disrupting ongoing operations.

Each DB cluster has a DB cluster identifier. This auto-generated name uniquely identifies the DB cluster when interacting with the Amazon Timestream for InfluxDB API and AWS CLI commands. The DB cluster identifier is unique for that customer in an AWS Region.

Timestream for InfluxDB allocates a DNS endpoint for your cluster. The service-generated identifier forms part of the DNS endpoint of the cluster. For example, if the cluster's service-generated identifier is `xghozx1v79`, then the DNS endpoint is `xghozx1v79-3ksj4dla5nfjhi.timestream-influxdb3.us-east-1.on.aws`.

Amazon Timestream for InfluxDB 3 allows you to create a master user account and password for your DB cluster as part of the creation process. This master user has permissions to create datababses, tables, and to perform read, write, delete and upsert operations on your data.

To access a visual, web-based user interface, you can use the InfluxDB Explore, which can be downloaded from InfluxData and run on your own instances.

DB Cluster Classes

The DB cluster class determines the computation and memory capacity of nodes within your Amazon Timestream for InfluxDB 3 clusters. The appropriate cluster class depends on your processing power and memory requirements.

InfluxDB 3 Cluster Classes

InfluxDB 3 clusters use instance classes specifically optimized for the Amazon S3-based storage architecture. These classes, designated as `db.influxIOIncluded`, are designed to balance compute resources with the I/O requirements of working with object storage. The `db.influxIOIncluded` compute instances are priced to already include all I/O operations performed against Amazon S3, providing customers with predictable costs regardless of their specific I/O patterns.

The following table shows the hardware details for InfluxDB 3 cluster node classes:

Instance Class	vCPU	Memory (GiB)	Storage Type	Network bandwidth (Gbps)
db.influxIOIncluded.medium	1	8	S3 Object Storage	12
db.influxIOIncluded.large	2	16	S3 Object Storage	12
db.influxIOIncluded.xlarge	4	32	S3 Object Storage	15
db.influxIOIncluded.2xlarge	8	64	S3 Object Storage	20
db.influxIOIncluded.4xlarge	16	128	S3 Object Storage	25
db.influxIOIncluded.8xlarge	32	256	S3 Object Storage	30
db.influxIOIncluded.12xlarge	48	384	S3 Object Storage	22.5
db.influxIOIncluded.16xlarge	64	512	S3 Object Storage	30

Instance Class	vCPU	Memory (GiB)	Storage Type	Network bandwidth (Gbps)
db.influxIOIncluded.24xlarge	96	768	S3 Object Storage	40

The `db.influxIOIncluded` instance classes are specifically designed to:

1. Efficiently manage the I/O operations between compute nodes and Amazon S3 object storage.
2. Provide optimal memory-to-CPU ratios for processing time-series data using the Apache Arrow framework.
3. Support the computational requirements of the Data Fusion query engine.

For Enterprise deployments with multi-node clusters, all nodes use the same instance class to ensure balanced performance across the cluster. As your workload grows, you can scale up by selecting a larger instance class for all nodes in the cluster or scale out by adding more nodes of the same class to your cluster.

Hardware specifications for DB cluster classes

The following terminology describes the hardware specifications for DB cluster classes:

- **vCPU** – The number of virtual central processing units (CPUs). A virtual CPU is a unit of capacity that you can use to compare DB cluster classes.
- **Memory (GiB)** – The RAM, in gibibytes, allocated to the DB cluster node. There is often a consistent ratio between memory and vCPU.
- **Storage Type** – InfluxDB 3 uses S3 Object Storage, which is separate from the compute nodes.
- **Network bandwidth** – The network speed relative to other DB cluster classes.

DB Cluster Storage

Amazon Timestream for InfluxDB 3 introduces a different storage architecture that decouples compute from storage by leveraging Amazon S3 for object storage. This architecture provides virtually unlimited storage capacity while keeping cost under control in large-scale deployments.

InfluxDB Object Storage

InfluxDB 3 clusters use a single storage class called InfluxDB Object Storage, which is built on Amazon S3. This storage option offers:

- Virtually unlimited storage capacity
- Cost-efficient storage for large data volumes (up to 75% savings for deployments exceeding 16TB compared to InfluxDB 2.x)
- Durability of 99.999999999% (11 nines)
- Data stored in the columnar Parquet format for efficient compression and query performance

Unlike traditional database architectures where storage is directly attached to compute nodes, InfluxDB 3's object storage is shared across all nodes in the cluster. Each node maintains local in-memory caches for frequently accessed data to optimize query performance while the underlying data remains in Amazon S3.

The InfluxDB Object Storage pricing is based on:

- Total data volume stored (GB/month), with a minimum monthly charge equivalent to 200GB

I/O operations costs are bundled within the compute pricing for the db.influxIOIncluded instance classes, providing customers with more predictable costs regardless of their specific I/O patterns.

Storage Architecture Benefits

The Amazon S3-based storage architecture in InfluxDB 3 is particularly beneficial for:

- Large-scale time-series analytics workloads
- High-cardinality data requirements
- Long-term data retention scenarios
- Cost-effective storage of large time-series datasets

This storage model is comparable to Timestream for LiveAnalytics' Magnetic Storage Tier, which is also object store based and optimized for long-term storage and analytics use cases. InfluxDB 3's storage economics are designed to be competitive with LiveAnalytics' magnetic store pricing, a model that has proven suitable for customers who want to run and store large-scale time-series workloads and data for long periods of time.

DB Cluster Sizing

The optimal configuration of a Timestream for InfluxDB 3 cluster depends on various factors, including ingestion rate, batch sizes, time-series cardinality, concurrent queries, and query types.

When sizing InfluxDB 3 clusters, consider these additional factors due to its different architecture:

- **Storage Efficiency**: For deployments with more than 16TB of data, InfluxDB 3's object storage can provide up to 75% cost savings compared to Timestream for InfluxDB 2 deployments.
- **Workload Characteristics**: InfluxDB 3 is optimized for high-cardinality data and analytical queries over large datasets while delivering comparable performance to versions 1.x and 2.x for shorter time-range queries. Additionally, features like the last-value and distinct-value caches enable sustained sub-10ms query latencies for specific use cases such as retrieving the most recent data points or unique metadata values.
- **Query Patterns**: Consider whether your workload is continuous (24/7 monitoring) or intermittent (periodic analytics)
- **Compaction Requirements**: Enterprise version includes compaction capabilities essential for long-term data storage and high-cardinality workloads. For deployments with 3-node clusters and larger, a dedicated compactor node is recommended to maximize writer and reader performance on the other individual nodes, ensuring that compaction processes don't compete for resources with query and ingestion operations.

To provide sizing recommendations, let's consider an exemplary workload with the following characteristics:

- Data is collected and written by a fleet of Telegraf agents gathering System, CPU, Memory, Disk, IO, etc. from a data center.
- Each write request contains 5000 lines.
- The queries executed on the system are categorized as "moderate complexity" queries, exhibiting the following characteristics:
  - They have multiple functions and one or two regular expressions.
  - They may include group by clauses or sample a time range of multiple weeks.
  - They typically take a few hundred milliseconds to a couple of thousand milliseconds to execute.
  - The CPU favors query performance primarily.
  - All tests were performed with a dataset containing a cardinality of 30 million time-series.

Writes (lines per second)	Reads (Queries per second)	Instance class	Version
~150,000	<25	db.influxIOIncluded.large	Core
~200,000	~25	db.influxIOIncluded.xlarge	Core
~250,000	~35	db.influxIOIncluded.2xlarge	Enterprise
~500,000	~50	db.influxIOIncluded.4xlarge	Enterprise
<750,000	<100	db.influxIOIncluded.8xlarge	Enterprise
>750,000	>100	Multi-node Enterprise cluster	Enterprise

DB Cluster Billing for Amazon Timestream for InfluxDB 3

Amazon Timestream for InfluxDB 3 clusters are billed based on the following components:

- **DB cluster node hours (per hour)** – Based on the DB cluster node class, for example, db.influxIOIncluded.large. Pricing is listed on a per-hour basis, but bills are calculated down to the second and show times in decimal form. Usage is billed in 1-second increments, with a minimum of 10 minutes. For single-node Core deployments or multi-node Enterprise deployments, each node is billed separately based on its running time.
- **Object storage (per GB per month)** – Actual data volume stored in InfluxDB Object Storage, rather than provisioned capacity. This model allows you to pay only for the storage you use, with no need to pre-provision storage capacity. **Note: There is a minimum monthly storage charge equivalent to 200GB**, even if your actual storage usage is lower.
- **Data transfer out (per GB)** – Data transfer out of your DB cluster to the internet or other AWS Regions. Data transfer in or traffic within the same VPC or between Availability Zones within the same AWS Region is free of charge.
- **InfluxDB 3 Enterprise License (per vCPU per hour)** – For InfluxDB 3 Enterprise deployments, there is an additional InfluxData license cost charged on a per vCPU per hour basis. This license fee is billed directly through AWS Marketplace and appears as a separate line item on your AWS bill. The license enables Enterprise features such as multi-node clustering, compaction capabilities, and advanced security features.

For InfluxDB 3 Enterprise clusters with multiple nodes, each node is billed separately based on its instance class and vCPU count (for license fees), while storage costs are consolidated since all nodes share the same underlying object storage.

Billing Components Summary

Billing Component	InfluxDB 3 Core	InfluxDB 3 Enterprise
Compute Billing	Per node hour (db.influxIOIncluded classes)	Per node hour (db.influxIOIncluded classes)
Storage Billing	Per GB of data stored (200GB minimum)	Per GB of data stored (200GB minimum)
I/O Operations	Included in compute node pricing	Included in compute node pricing
Storage Scaling	Not necessary,pay only for storage used	Not necessary, pay only for storage used
Data Transfer In	Free	Free
Data Transfer Within VPC/AZ	Free	Free
Data Transfer Out (Internet/Cross-Region)	Charged per GB	Charged per GB
Minimum Billing	10 minutes	10 minutes
Billing Granularity	1-second increments	1-second increments
Multi-Node Support	No (single-node only)	Yes
Additional License	None	Per vCPU per hour via AWS Marketplace

Cost Optimization Strategies

- **Right-sizing** – Select the appropriate node class based on your workload requirements to avoid over-provisioning. We also recommend choosing an instance that maintains an average of approximately 65% CPU and memory utilization, ensuring resilience to usage spikes and allowing headroom for healthy capacity planning.
- **Storage Efficiency** – For deployments with more than 16TB of data, InfluxDB 3's object storage can provide up to 75% cost savings compared to traditional EBS-based solutions.
- **Network Traffic Optimization** – Keep data transfer within the same VPC or region when possible to avoid data transfer out charges.
- **Scaling Strategy** – For InfluxDB 3 Enterprise, consider whether scaling up (larger node class) or scaling out (more nodes) is more cost-effective for your specific workload patterns.
- **License Optimization** – For InfluxDB 3 Enterprise, choose node classes that provide the optimal balance between vCPU count (which affects license costs) and performance requirements.

For Amazon Timestream for InfluxDB 3 pricing information, see the Amazon Timestream for InfluxDB pricing page.

AWS Regions and Availability Zones

Amazon cloud computing resources are hosted in multiple locations world-wide. These locations are composed of AWS Regions and Availability Zones. Each AWS Region is a separate geographic area. Each AWS Region has multiple, isolated locations known as Availability Zones.

Amazon Timestream for InfluxDB 3 enables you to place resources, such as DB clusters, and data in multiple locations. For Enterprise deployments with multi-node clusters, nodes are distributed across multiple Availability Zones to enhance availability.

For information about AWS Regions where Amazon Timestream for InfluxDB 3 is available and the endpoints for each Region, see Amazon Timestream endpoints and quotas.

Getting started with Timestream for InfluxDB 3

Setting up Amazon Timestream for InfluxDB 3

Before you use Amazon Timestream for InfluxDB for the first time, complete the following tasks:

If you already have an AWS account, know your Timestream for InfluxDB requirements, and prefer to use the defaults for IAM and Amazon VPC, skip to Creating and connecting to a Timestream for InfluxDB instance.

Determine requirements

Before you create a DB instance and a security group, you must know your DB instance and network needs. Here are some important things to consider:

- **Resource requirements** – What are the memory and processor requirements for your application or service?
- **VPC and security group** – Your DB instance will most likely be in a virtual private cloud (VPC). To connect to your DB instance, you need to set up security group rules.
- **High availability** – Do you need failover support?
- **IAM policies** – Does your AWS account have policies that grant the permissions needed to perform Amazon Timestream for InfluxDB operations?
- **Open ports** – What TCP/IP port does your database listen on? The default for Timestream for InfluxDB is 8086.
- **AWS Region** – What AWS Region do you want your database in?
- **DB disk subsystem** – What are your storage requirements?

Provide access to your DB instance in your VPC by creating a security group

VPC security groups provide access to DB instances in a VPC. They act as a firewall for the associated DB instance, controlling both inbound and outbound traffic at the DB instance level. DB instances are created by default with a firewall and a default security group that protect the DB instance.

To create a VPC security group:

1. In the AWS Management Console, choose **VPC**.
2. In the navigation pane, choose **Security Groups**.
3. Choose **Create security group**.
4. Enter a name, description, and select your VPC.
5. Add inbound rules for Custom TCP with appropriate source settings.
6. Create the security group.

Creating and connecting to a Timestream for InfluxDB instance

Connecting to a Amazon Timestream for InfluxDB DB instance uses token authentication.

The connection information includes endpoint, port, username, password, and a valid access token. You can find this information using the AWS Management Console or AWS CLI.

You can create access tokens using:

- The InfluxDB CLI
- The InfluxDB 3 Explorer
- The InfluxDB API

The following procedure creates both an Amazon Elastic Compute Cloud instance and a Timestream for InfluxDB DB cluster, and shows you how to write data to the DB instance from the Amazon EC2 instance using the Telegraf client.

Step 1: Create an Amazon EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console.
2. Choose the AWS Region where you want to create the Amazon EC2 instance.
3. Choose Amazon EC2 Dashboard, then Launch instance.
4. Configure your Amazon EC2 instance with appropriate settings.

Step 2: Create an InfluxDB 3 instance

1. Sign in to the AWS Management Console and open the Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**.
3. Choose **Create InfluxDB 3 database**.
4. After selecting InfluxDB 3, choose between the Core and Enterprise editions. For this tutorial, where you'll be ingesting data from a single Amazon EC2 instance and running test queries, the Core edition is sufficient for your needs.
5. Configure your DB instance with appropriate settings. For specific engine configurations, you can select from an existing parameter group or create a new one. If no custom configuration is needed, simply proceed and a default parameter group will be automatically created for your instance.

Important: If creating a private cluster, make sure you add the required S3 policies to create your Timestream for InfluxDB cluster

6. Configure your instance size and network settings. Pay special attention to the network configuration. If you choose a private instance, ensure it's accessible from your Amazon EC2 instance's VPC by selecting the appropriate VPC, subnets, and security groups that allow connectivity between your Amazon EC2 instance and the InfluxDB instance.
7. Choose **Create InfluxDB database**.
8. Wait for your DB instance to become available.

Step 3: Access the InfluxDB Explorer

The easiest way to start interacting with your InfluxDB instance through the InfluxDB Explorer:

1. Download the InfluxDB Explorer from https://docs.influxdata.com/influxdb3/explorer/
2. For private DB instances, run the Explorer from within the same VPC (using an Amazon EC2 instance or bastion host).
3. For publicly accessible DB instances, you can run the Explorer from any location with internet access.
4. Configure the Explorer with your cluster endpoint and credentials.

If you prefer using the InfluxDB 3 CLI, or APIs, please refer to InfluxDB 3 documentation to find information on writing data, executing queries, or Administer your InfluxDB 3 Database

Step 4: Send Telegraf data to your InfluxDB instance

1. Connect to your InfluxDB instance using the InfluxDB Explorer and generate an API token.
2. Connect to your Amazon EC2 instance and install Telegraf.
3. Configure Telegraf to send data to your InfluxDB instance.
4. Enable and start the Telegraf service.

Step 5: Delete the Amazon EC2 instance and the InfluxDB DB instance

After you explore the Telegraf-generated data, delete both your Amazon EC2 and your InfluxDB DB instances to avoid being charged for them.

Configuring an InfluxDB 3 cluster in Timestream

Creating a DB cluster in Timestream

Using the AWS Management Console:

1. Open the Amazon Timestream for InfluxDB console.
2. Choose **InfluxDB Databases** in the navigation pane.
3. Choose **Create InfluxDB 3 database.**
4. Select **InfluxDB 3** as your engine version.
5. Choose **Enterprise** edition for production workloads.
6. Configure cluster deployment:
   a. Select either **1-node** (single node handling writer, reader, and compactor roles), **3-node** configuration, or a **multi-node cluster (up to 15 nodes)** using a custom parameter group. For details on multi-node configurations, see the section called "Scaling a cluster".

   > **Note**
   > Single node deployments use Single-AZ compute resources. In case of an issue, a new compute resource must be initialized, resulting in longer remediation time

   b. For multi-node deployments, the system configures:
      i. 1-4 writer/reader nodes for handling both write and read operations.
      ii. 0-13 reader-only nodes dedicated to processing read queries.
      iii. One dedicated compactor node for storage optimization (required for clusters with 3+ nodes).
7. Configure cluster-level settings:
   a. **Instance class**: Select the appropriate `db.influx` instance size (applies to all nodes). Worth noting your bill will show `db.influxIOIncluded` on your records.
   b. **Parameter group**: Choose an existing parameter group or create a new one for custom engine configuration.
   c. **Network configuration**: Configure VPC, subnets, and security groups (applies to all nodes).

> **Note**
>
> Shared VPCs are not currently supported for Timestream for InfluxDB 3.

d. **Public accessibility**: Choose whether the cluster endpoints should be publicly accessible.

8. **Important:** If creating a private cluster, make sure you add the required S3 policies to create your Timestream for InfluxDB 3 cluster
9. Review your configuration and choose **Create InfluxDB database**

Using the AWS CLI:

```
aws timestream-influxdb create-db-cluster \
     --name myinfluxDbinstance \
     --db-instance-type db.influx.4xlarge \
     --vpc-subnet-ids subnetid1 subnetid2 \
     --vpc-security-group-ids mysecuritygroup \
     --db-parameter-group-identifier dbparametergroupidentifier
```

> **Note**
>
> You can label Timestream for InfluxDB resources using tags. Tags let you categorize your resources in different ways, such as by purpose, owner, environment, or other criteria.

Configure S3 VPC Endpoint for Private Clusters

When deploying private clusters, you must configure an S3 VPC endpoint with appropriate permissions to ensure cluster resources can access required S3 buckets.

> **Note**
>
> The S3 endpoint should preferably be associated with a VPC that hosts only Timestream for InfluxDB private databases.

> **Note**
>
> Shared VPCs are not currently supported for Timestream for InfluxDB 3.

Prerequisites

- A VPC with private subnets configured for your cluster
- Appropriate IAM permissions to create and modify VPC endpoints

Required S3 Endpoint Policy

Your S3 VPC endpoint requires a policy that grants sufficient access for cluster operations. The following example provides full S3 access through the endpoint:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "FullAccess",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "*",
            "Resource": "*"
        }
    ]
}
```

Configuration Steps

1. Navigate to the VPC console and select **Endpoints**
2. Choose your S3 endpoint or create a new one
3. In the **Policy** tab, replace the existing policy with the example above
4. Save your changes

Settings for DB clusters

Key settings that apply to all nodes in your cluster include:

- **Cluster identifier**: Unique name for your cluster
- **Instance class**: Compute and memory capacity (same for all nodes)
- **Node configuration**: 1-node, 3-node, or multi-node deployment (up to 15 nodes)
- **Parameter group**: Engine configuration settings
- **Network type**: IPv4 or dual-stack
- **VPC and subnets**: Network isolation and availability
- **Security groups**: Network access control
- **Public accessibility**: Internet connectivity option

Important Considerations

- **Uniform node configuration**: All nodes in a cluster must use the same instance class and network configuration.
- **Scalability**: Enterprise clusters support up to 15 nodes. You can scale your cluster by creating a new parameter group with your desired node configuration and applying it to your cluster. See the section called “Scaling a cluster”.
- **High availability**: Multi-node configurations provide better availability and performance distribution, with nodes spread across multiple Availability Zones.
- **Compactor optimization**: In multi-node deployments (3+ nodes), the dedicated compactor node ensures write and read performance isn't impacted by background optimization tasks

Topics

- Parameter Groups for DB Clusters in Amazon Timestream
- Core and Enterprise versions in Timestream for InfluxDB 3
- Deployment models in Timestream for InfluxDB 3
- Endpoints and connectivity for Timestream for InfluxDB 3

Parameter Groups for DB Clusters in Amazon Timestream

Database parameters specify how the database is configured. You manage your database configuration by associating your DB instances with parameter groups.

Timestream defines parameter groups with default settings. You can also define your own parameter groups with customized settings. **Parameter groups for Core and Enterprise editions, while similar, are not identical or interchangeable.**

For InfluxDB 3, cluster configurations are managed through parameter groups. These parameter groups contain engine configuration values that determine how your InfluxDB 3 cluster operates.

Important: All parameters are startup-only. There is no runtime reconfiguration mechanism. To change any parameter, the node/cluster must be restarted with updated arguments.

Topics

- Parameter Group Characteristics
- Creating Parameter Groups with the AWS CLI
- Supported Instance Types and Specifications
- Detailed Parameter Reference

Parameter Group Characteristics

Parameter Group Characteristics

Characteristic	Description
Immutability	Parameter groups are immutable once created. You cannot modify the parameters after creation.
Cloning	You can create a new parameter group by cloning an existing one and modifying the parameters during the cloning process. This is a AWS Management Console only feature. **Note:** When cloning, `gen1-duration` , `compaction-gen2-duration` ,

Characteristic	Description
	`compaction-multipliers` , and `gen1-lookback-duration` must be copied unchanged — see the section called "Detailed Parameter Reference" for details on parameter s that must not change after initial setup.
Deletion	Parameter groups cannot be deleted.
Default Groups	Amazon Timestream provides default parameter groups with pre-configured settings optimized for common use cases.
Edition Specificity	Enterprise parameter groups can only be used with Enterprise clusters; Core parameter groups can only be used with Core clusters.

Creating Parameter Groups with the AWS CLI

You can create custom parameter groups using the AWS CLI to configure your InfluxDB 3 clusters. The following examples demonstrate how to create parameter groups with commonly modified parameters.

Example: Creating an Enterprise parameter group with custom query settings:

```
aws timestream-influxdb create-db-parameter-group \
  --name "my-enterprise-pg" \
  --description "Custom Enterprise parameter group" \
  --parameters '{
    "InfluxDBv3Enterprise": {
      "queryFileLimit": 500,
      "queryLogSize": 2000
    }
  }'
```

Example: Creating a Core parameter group with custom DataFusion threading:

```
aws timestream-influxdb create-db-parameter-group \
```

```
  --name "my-core-pg" \
  --description "Custom Core parameter group" \
  --parameters '{
    "InfluxDBv3Core": {
      "dataFusionNumThreads": 32,
      "queryFileLimit": 600
    }
  }'
```

Note: After creating a parameter group, you can use it when creating a new DB cluster by specifying `--db-parameter-group-identifier`, or when updating an existing cluster using the `update-db-cluster` command. Parameter groups cannot be modified after creation.

Verifying your current running configuration: You can check the effective parameter values on a running cluster by querying the `_internal` database's `nodes` table:

```
SELECT * FROM _internal.nodes
```

This query returns the active configuration for each node in your cluster, allowing you to verify parameter values before and after applying a new parameter group.

Supported Instance Types and Specifications

The following table lists all supported instance types for Amazon Timestream, along with their hardware specifications. These specifications are critical for determining the optimal parameter group values for your workload.

Supported Instance Types

Instance Type	vCPUs	Memory (GiB)	Memory (Bytes)	Use Case Profile
db.influx.medium	1	8	8,589,934,592	Dev/test, low-volume workloads
db.influx.large	2	16	17,179,869,184	Light production, small datasets
db.influx.xlarge	4	32	34,359,738,368	Small production workloads

Instance Type	vCPUs	Memory (GiB)	Memory (Bytes)	Use Case Profile
db.influx.2xlarge	8	64	68,719,476,736	Medium production workloads
db.influx.4xlarge	16	128	137,438,953,472	Large production workloads
db.influx.8xlarge	32	256	274,877,906,944	High-throughput production
db.influx.12xlarge	48	384	412,316,860,416	Heavy analytics workloads
db.influx.16xlarge	64	512	549,755,813,888	Maximum r7g capacity
db.influx.24xlarge	96	768	824,633,720,832	Extreme-scale workloads

The following table provides recommended parameter values for each instance size. Parameters not listed use the same default across all sizes. For detailed explanations, see the section called "Detailed Parameter Reference".

Recommended Parameter Values by Instance Size

Paramet	Medium	Large	XLarge	2XLarge	4XLarge	8XLarge	12XLarg	16XLarg	24XLarge
num-datafusion-threads	1	2	4	8	16	32	48	64	96

Paramet	Medium	Large	XLarge	2XLarge	4XLarge	8XLarge	12XLarg	16XLarg	24XLarge
num-io-threads	1	2	4	8	16	32	48	64	96
datafusion-max-parquet-fanout	250–500	500	500–1K	1K	1K–2K	2K–5K	5K–10K	5K–10K	5K–10K
query-file-limit	100–200	200–300	300–432	432	432–600	600–800	800–1024	800–1024	1024
query-log-size	500	1K	1K	1K–2K	2K–5K	2K–5K	2K–5K	2K–5K	2K–5K
exec-mem-pool-bytes	20%	20%	20%	20%	20%	20%	20%	20%	20%
force-snapshot-mem-threshold	30%	30%	30%	30%	30%	30%	30%	30%	30%
parquet-mem-cache-size	15%	20%	20%	20%	25%	25%	25%	25%	25%

Paramet	Medium	Large	XLarge	2XLarge	4XLarge	8XLarge	12XLarg	16XLarg	24XLarge
wal-max-write-buffer-size	50K–100K	100K	100K–200K	200K–300K	300K–500K	500K–700K	500K–800K	700K–1M	800K–1M
wal-snapshot-size	300	300	300	300	300	300	300	300	300
compactn-max-num-files-per-plan (Ent.)	500	500	500	500–1K	1K–2K	2K–5K	5K	5K–10K	5K–10K
table-index-cache-concurrency-limit	8–16	16–20	20	20–24	24–32	32–48	48–64	48–64	64–100
max-http-request-size	5 MiB	10 MiB	10 MiB	10–16 MiB	10–16 MiB	10–16 MiB	10–16 MiB	10–16 MiB	10–16 MiB

Note: Parameters not listed in this table (such as `datafusion-use-cached-parquet-loader`, `disable-parquet-mem-cache`, `parquet-mem-cache-prune-interval`, `compaction-multipliers`, `compaction-row-limit`, cache eviction intervals, and others) use the same

recommended default value across all instance sizes. See the section called "Detailed Parameter Reference" for their defaults.

Note: Parameters not listed in the table above use the same recommended default value across all instance sizes. See the section called "Detailed Parameter Reference" for their defaults.

Detailed Parameter Reference

The following parameters are available for InfluxDB 3 parameter groups. Most parameters apply to both Core and Enterprise editions. Parameters marked **Enterprise only** are not available in Core.

> **Note**
>
> **Parameters that must not change after initial setup:** The parameters `gen1-duration`, `compaction-gen2-duration`, `compaction-multipliers`, and `gen1-lookback-duration` are persisted in the catalog. Changing them after the cluster's first start causes silent data divergence (deduplication failures, incorrect generation durations). Set these at cluster creation time only. When cloning parameter groups, these values must be copied unchanged.

Parameter Summary by Edition

Parameter	Category	Edition
`num-datafusion-threads`	Query Execution	Core, Enterprise
`num-io-threads`	Query Execution	Core, Enterprise
`datafusion-max-parquet-fanout`	Query Execution	Core, Enterprise
`datafusion-use-cached-parquet-loader`	Query Execution	Core, Enterprise
`datafusion-config`	Query Execution	Core, Enterprise
`query-file-limit`	Query Execution	Core, Enterprise

Parameter	Category	Edition
query-log-size	Query Execution	Core, Enterprise
exec-mem-pool-bytes	Memory Management	Core, Enterprise
force-snapshot-mem-threshold	Memory Management	Core, Enterprise
parquet-mem-cache-size	Memory Management	Core, Enterprise
disable-parquet-mem-cache	Memory Management	Core, Enterprise
parquet-mem-cache-prune-interval	Memory Management	Core, Enterprise
parquet-mem-cache-prune-percentage	Memory Management	Core, Enterprise
parquet-mem-cache-query-path-duration	Memory Management	Core, Enterprise
preemptive-cache-age	Memory Management	Core, Enterprise
wal-max-write-buffer-size	WAL Configuration	Core, Enterprise
wal-snapshot-size	WAL Configuration	Core, Enterprise
snapshotted-wal-files-to-keep	WAL Configuration	Core, Enterprise
compaction-check-interval	Compaction	Enterprise only
compaction-cleanup-wait	Compaction	Enterprise only

Parameter	Category	Edition
compaction-gen2-duration	Compaction	Enterprise only
compaction-max-num-files-per-plan	Compaction	Enterprise only
compaction-multipliers	Compaction	Enterprise only
compaction-row-limit	Compaction	Enterprise only
gen1-duration	Data Lifecycle	Core, Enterprise
gen1-lookback-duration	Data Lifecycle	Core, Enterprise
delete-grace-period	Data Lifecycle	Core, Enterprise
hard-delete-default-duration	Data Lifecycle	Core, Enterprise
retention-check-interval	Data Lifecycle	Core, Enterprise
distinct-cache-eviction-interval	Caching	Core, Enterprise
distinct-value-cache-disable-from-history	Caching	Enterprise only
last-cache-eviction-interval	Caching	Core, Enterprise
last-value-cache-disable-from-history	Caching	Enterprise only

Parameter	Category	Edition
`table-index-cache-concurrency-limit`	Table Index Cache	Core, Enterprise
`table-index-cache-max-entries`	Table Index Cache	Core, Enterprise
`max-http-request-size`	HTTP and Network	Core, Enterprise
`log-filter`	Logging	Core, Enterprise
`catalog-sync-interval`	Catalog and Replication	Enterprise only
`replication-interval`	Catalog and Replication	Enterprise only

Category 1: Query Execution and DataFusion Engine

These parameters control the Apache DataFusion query engine, which is responsible for executing SQL and InfluxQL queries against your data. Proper tuning of these parameters directly impacts query latency, throughput, and concurrent query capacity.

`num-datafusion-threads`

Parameter Details

Default	System logical core count (number of vCPUs)
Allowed Values	Integer: 1 – 2,048
Category	Query Execution

Detailed Explanation:

Sets the number of worker threads that the DataFusion query engine uses for parallel query execution. Each thread can independently process query partitions, enabling parallelism within a single query as well as across multiple concurrent queries.

Impact:

- **Too low:** Queries execute serially or with minimal parallelism, leading to high query latency. CPU resources remain underutilized.
- **Too high:** Excessive thread contention, context switching overhead, and potential memory pressure.
- **Optimal: Set to the number of available vCPUs.** If you are using read-only nodes you can assign more than 1 thread per vCPU, but we recommend extensive testing.

Recommendation: Set to the number of vCPUs on your instance.

`num-io-threads`

Parameter Details

Default	System logical core count (number of vCPUs)
Allowed Values	Integer: 1 – 2,048
Category	Query Execution / I/O

Detailed Explanation:

Sets the number of threads in the I/O runtime, which handles network I/O, object store operations, and other async I/O tasks. This is separate from the DataFusion query threads.

Recommendation: Set to the number of vCPUs on your instance.

`datafusion-max-parquet-fanout`

Parameter Details

Default	1000
Allowed Values	Integer: 1 – 1,000,000
Category	Query Execution

Detailed Explanation:

Controls the maximum number of Parquet files that can be read in parallel during a single query operation. A high fanout value does not mean all files are read simultaneously — the thread pool still controls actual parallelism. The fanout value controls how the query plan is structured.

Recommendations by Instance Size:

Instance Size Recommendations

Instance Type	vCPUs	Memory (GiB)	Recommended Value
db.influx.medium	1	8	**250–500**
db.influx.large	2	16	**500**
db.influx.xlarge	4	32	**500–1000**
db.influx.2xlarge	8	64	**1000**
db.influx.4xlarge	16	128	**1000–2000**
db.influx.8xlarge	32	256	**2000–5000**
db.influx.12xlarge+	48+	384+	**5000–10000**

`datafusion-use-cached-parquet-loader`

Parameter Details

Default	TRUE
Allowed Values	FALSE, TRUE
Category	Query Execution

Detailed Explanation:

When enabled, DataFusion uses a cached Parquet loader that keeps parsed Parquet file metadata and footer information in memory, avoiding re-parsing on subsequent reads.

Recommendation: Keep as TRUE (default) for all instance sizes. Set to FALSE only if memory is extremely constrained.

`datafusion-config`

Parameter Details

Default	*(empty)*
Allowed Values	Colon-separated key:value pairs (e.g., `batch_size:16384,coalesce_batches:true`)
Category	Query Execution

Detailed Explanation:

Allows passing additional key-value configuration parameters directly to the DataFusion engine. **Format is comma-separated `key:value` pairs using colons, not equals signs.**

Recommendation: Leave empty (default) unless you have specific DataFusion tuning requirements validated through testing.

`query-file-limit`

Parameter Details

Default	432
Allowed Values	Integer: 0 – 1,024
Category	Query Execution

Detailed Explanation:

Sets the maximum number of Parquet files that can be included in a single query's execution plan. If a query would need to scan more files than this limit, it is rejected. A value of 0 disables the limit.

Recommendations by Instance Size:

Instance Size Recommendations

Instance Type	Memory (GiB)	Recommended Value
db.influx.medium	8	**100–200**
db.influx.large	16	**200–300**
db.influx.xlarge	32	**300–432**
db.influx.2xlarge	64	**432**
db.influx.4xlarge	128	**432–600**
db.influx.8xlarge+	256+	**600–1024**

`query-log-size`

Parameter Details

Default	1000
Allowed Values	Integer: 1 – 10,000
Category	Query Execution / Observability

Detailed Explanation:

Controls the maximum number of entries retained in the in-memory query log for debugging and performance analysis.

Recommendation: 500 for db.influx.medium, 1000 (default) for most instances, 2000–5000 for db.influx.4xlarge and above.

Category 2: Memory Management

These parameters control how memory is allocated across query execution, caching, and data management. Proper memory tuning is critical — InfluxDB 3 is memory-intensive, and misconfiguration can lead to out-of-memory conditions or underutilization.

`exec-mem-pool-bytes`

Parameter Details

Default	`20%` of system memory
Allowed Values	Percentage (e.g., `70%`) or absolute number (0 – 1,610,612,736,000)
Category	Memory Management

Detailed Explanation:

Defines the maximum amount of memory that the query execution engine (DataFusion) can use for processing queries. This includes memory for sorting, aggregation, joins, and intermediate result sets.

Recommendation: Keep at `20%` (default) for all instance sizes. For query/reader-only nodes, you can increase up to 70%.

`force-snapshot-mem-threshold`

Parameter Details

Default	`50%`
Allowed Values	Percentage (e.g., `70%`) or absolute number (0 – 1,610,612,736,000)
Category	Memory Management

Detailed Explanation:

Defines the memory usage threshold at which the system forces a WAL snapshot to disk. When memory usage exceeds this threshold, in-memory WAL data is persisted to Parquet files, freeing memory.

Recommendation: 30% for all instance sizes.

`parquet-mem-cache-size`

Parameter Details

Default	`20%` of system memory
Allowed Values	Percentage (e.g., `20%`) or absolute number (0 – 1,610,612,736,000)
Category	Memory Management / Caching

Detailed Explanation:

Sets the maximum amount of memory dedicated to caching Parquet file data in memory. This cache stores recently accessed Parquet data blocks, dramatically reducing read latency for repeated queries.

Recommendation: 15% for db.influx.medium, 20% for db.influx.large through db.influx.2xlarge, 25% for db.influx.4xlarge and above.

`disable-parquet-mem-cache`

Parameter Details

Default	FALSE
Allowed Values	FALSE, TRUE
Category	Memory Management / Caching

Detailed Explanation:

When set to TRUE, completely disables the Parquet memory cache. All Parquet data reads go directly to object storage.

Recommendation: Keep as FALSE (default). Only set to TRUE for dedicated write-only ingestion nodes.

`parquet-mem-cache-prune-interval`

Parameter Details

Default	1 second
Allowed Values	Duration
Category	Memory Management / Caching

Detailed Explanation:

Controls how frequently the system checks whether the Parquet memory cache has exceeded its size limit and needs pruning.

Recommendation: Keep at 1 second (default) for all instance sizes.

`parquet-mem-cache-prune-percentage`

Parameter Details

Default	0.1
Allowed Values	Float: 0 – 1
Category	Memory Management / Caching

Detailed Explanation:

When the Parquet memory cache is pruned, this parameter controls what percentage of the cache is evicted. A value of 0.1 means 10% of the cache is evicted during each prune cycle.

Recommendation: Keep at 0.1 (default) for all instance sizes.

`parquet-mem-cache-query-path-duration`

Parameter Details

Default	5 hours
Allowed Values	Duration

Category	Memory Management / Caching

Detailed Explanation:

Controls how long query access path information is retained for Parquet cache entries. This metadata helps the cache make intelligent eviction decisions.

Recommendation: Keep at 5 hours (default). Increase to 10–15 hours for periodic queries on infrequent schedules.

`preemptive-cache-age`

Parameter Details

Default	3 days
Allowed Values	Duration
Category	Memory Management / Caching

Detailed Explanation:

Controls the age threshold for preemptive cache warming after compaction. When new compacted files newer than this age are created, they are preemptively loaded into the Parquet memory cache.

Recommendation: Keep at 3 days (default). Reduce if cache memory pressure is an issue; increase if queries frequently access data older than 3 days.

Category 3: Write-Ahead Log (WAL) Configuration

The WAL ensures data durability during ingestion. Data is first written to the WAL before being persisted to Parquet files. These parameters control WAL behavior, snapshot creation, and flush characteristics.

`wal-max-write-buffer-size`

Parameter Details

Default	100,000
Allowed Values	Integer: 1 – 1,000,000

Category	WAL / Ingestion

Detailed Explanation:

Sets the maximum number of write operations that can be buffered in the WAL write buffer before being flushed.

Recommendations by Instance Size:

Instance Size Recommendations

Instance Type	Memory (GiB)	Recommended Value
db.influx.medium	8	**50,000–100,000**
db.influx.large	16	**100,000**
db.influx.xlarge	32	**100,000–200,000**
db.influx.2xlarge	64	**200,000–300,000**
db.influx.4xlarge	128	**300,000–500,000**
db.influx.8xlarge+	256+	**500,000–1,000,000**

`wal-snapshot-size`

Parameter Details

Default	600
Allowed Values	Integer: 1 – 10,000
Category	WAL / Ingestion

Detailed Explanation:

Controls the size threshold (in number of WAL operations) at which a WAL snapshot is triggered, persisting data to Parquet files.

Recommendation: 300 for all instance sizes.

`snapshotted-wal-files-to-keep`

Parameter Details

Default	300
Allowed Values	Integer: 1 – 10,000
Category	WAL / Durability

> **Note**
>
> This value must always be non-zero. Replicas pull WAL data from snapshotted WAL files. Setting this to 0 would prevent replicas from catching up after a snapshot.

Detailed Explanation:

Controls the number of already-snapshotted WAL files to retain on disk. These files have been persisted to Parquet but are kept so that replicas can pull the WAL data. Essential for replication in Enterprise clusters.

Recommendation: Keep at 300 (default). Increase to 500–1000 for mission-critical workloads where replicas may experience extended outages.

Category 4: Compaction

Compaction parameters are available in the **Enterprise edition only**. Core does not include compaction capabilities. These parameters control how InfluxDB 3 optimizes Parquet files over time.

`compaction-check-interval`

Parameter Details

Default	10 seconds
Allowed Values	Duration

Category	Compaction

Detailed Explanation:

Controls how frequently the compactor evaluates whether compaction work is needed.

Recommendation: 10 seconds (default) for most instances. Reduce to 5 seconds for db.influx.4xlarge and above with high ingestion rates.

`compaction-cleanup-wait`

Parameter Details

Default	1 hour
Allowed Values	Duration
Category	Compaction

Detailed Explanation:

After compaction merges source files into a new compacted file, the original source files are not deleted immediately. This parameter defines the grace period before cleanup. The delay exists because in-flight queries may still hold references to the old files.

Recommendation: 30 minutes for short queries only. 1 hour (default) for mixed workloads. 1–2 hours for long analytical queries (30+ minutes).

`compaction-gen2-duration`

Parameter Details

Default	20 minutes
Allowed Values	Duration
Category	Compaction

> **Note**
>
> **IMMUTABLE AFTER INITIAL SETUP:** This parameter is persisted in the catalog. Changing it after the cluster's first start causes silent data divergence. Set this at cluster creation time only. When cloning parameter groups, this value must be copied unchanged.

Detailed Explanation:

Defines the time span that each Gen2 compacted file should cover. InfluxDB 3 uses a tiered compaction strategy: Gen0 (raw WAL snapshots), Gen1 (controlled by `gen1-duration`), Gen2 (this parameter), Gen3+ (controlled by `compaction-multipliers`).

Recommendation: 20 minutes (default) for real-time monitoring. 1 hour for operational dashboards. 2–4 hours for analytical/batch queries.

`compaction-max-num-files-per-plan`

Parameter Details

Default	500
Allowed Values	Integer: 1 – 10,000
Category	Compaction

Detailed Explanation:

Limits the maximum number of source files included in a single compaction plan.

Recommendation: 500 (default) for db.influx.medium through db.influx.xlarge. 500–1000 for db.influx.2xlarge. 1000–5000 for db.influx.4xlarge through db.influx.8xlarge. 5000–10000 for db.influx.12xlarge and above.

`compaction-multipliers`

Parameter Details

Default	`3,4,6,5`

Allowed Values	Four comma-separated integers
Category	Compaction

> **Note**
>
> **IMMUTABLE AFTER INITIAL SETUP:** This parameter is persisted in the catalog. Changing it after the cluster's first start affects all higher generation durations and causes silent data divergence. Set this at cluster creation time only. When cloning parameter groups, this value must be copied unchanged.

Detailed Explanation:

Defines the size multipliers for each compaction generation level beyond Gen2. With the default `compaction-gen2-duration` of 20 minutes and multipliers of `3,4,6,5`: Gen2 = 20m, Gen3 = 20m × 3 = 1h, Gen4 = 1h × 4 = 4h, Gen5 = 4h × 6 = 24h, Gen6 = 24h × 5 = 5d.

Recommendation: Keep at `3,4,6,5` (default) for all instance sizes.

`compaction-row-limit`

Parameter Details

Default	1,000,000
Allowed Values	Integer: 1 – 100,000,000
Category	Compaction

Detailed Explanation:

Sets the maximum number of rows that a single compaction operation can process. This is a memory safety mechanism — each row being compacted must be held in memory during the merge operation.

Recommendation: Keep at 1,000,000 (default) for all instance sizes.

Category 5: Data Lifecycle and Retention

`gen1-duration`

Parameter Details

Default	10 minutes
Allowed Values	Duration
Category	Data Lifecycle

> **Note**
>
> **IMMUTABLE AFTER INITIAL SETUP:** This parameter is persisted in the catalog. Changing it after the cluster's first start causes silent deduplication failures. Set this at cluster creation time only. When cloning parameter groups, this value must be copied unchanged.

Detailed Explanation:

Controls the time span covered by Gen1 Parquet files. Data flows from WAL to Gen1 to Gen2 (via compaction) to Gen3+.

Recommendation: 5–10 minutes for high-frequency ingestion. 10 minutes (default) for medium-frequency. 15–30 minutes for low-frequency. 30–60 minutes for batch ingestion.

`gen1-lookback-duration`

Parameter Details

Default	1 month
Allowed Values	Duration
Category	Data Lifecycle

> **Note**
>
> Leave at the default value (1 month) in almost all cases. Setting it to a small value can cause the cluster to not initialize with the correct historical state. Only increase this value (never decrease).

Detailed Explanation:

Defines how far back the system looks to correctly place data into the appropriate Gen1 time partition and to reconstruct the correct historical state during initialization.

Recommendation: Leave at 1 month (default) for all deployments.

`delete-grace-period`

Parameter Details

Default	24 hours
Allowed Values	Duration
Category	Data Lifecycle

Detailed Explanation:

When data is marked for deletion, this parameter defines the grace period before the deletion is physically applied. During this period, the data remains queryable (soft delete).

Recommendation: 15 minutes for dev/test. 1 hour for standard production. 4–24 hours for compliance-sensitive environments.

`hard-delete-default-duration`

Parameter Details

Default	72 hours (3 days)
Allowed Values	Duration

Category	Data Lifecycle

Detailed Explanation:

After the delete-grace-period expires and data is soft-deleted, this parameter controls how long the physical data remains on storage before being permanently removed (hard deleted).

Recommendation: 72 hours (default) for cost-sensitive workloads. 72 hours – 7 days for standard production. 30–90 days for compliance requirements.

`retention-check-interval`

Parameter Details

Default	30 minutes
Allowed Values	Duration
Category	Data Lifecycle

Detailed Explanation:

Controls how frequently the system evaluates and enforces retention policies.

Recommendation: 10–15 minutes for strict retention compliance. 30 minutes (default) for standard production. 1–6 hours for relaxed retention.

Category 6: Caching — Distinct Value and Last Value

`distinct-cache-eviction-interval`

Parameter Details

Default	10 seconds
Allowed Values	Duration
Category	Caching

Detailed Explanation:

The distinct value cache stores the set of unique values for tag columns. This parameter controls how frequently the cache is checked for stale entries.

Recommendation: Keep at 10 seconds (default) for all instance sizes.

`distinct-value-cache-disable-from-history` *(Enterprise only)*

Parameter Details

Default	FALSE
Allowed Values	FALSE, TRUE
Category	Caching (Enterprise only)

Detailed Explanation:

Controls whether the distinct value cache is populated from historical Parquet files in addition to real-time data. When FALSE, the system scans historical files for a complete picture of all distinct tag values.

Recommendation: FALSE (default) for query-heavy workloads. TRUE for write-heavy workloads or very high tag cardinality (millions).

`last-cache-eviction-interval`

Parameter Details

Default	10 seconds
Allowed Values	Duration
Category	Caching

Detailed Explanation:

The last value cache stores the most recent data point for each unique time series. This parameter controls how frequently stale entries are evicted.

Recommendation: Keep at 10 seconds (default). Reduce to 5 seconds if you have high time series churn.

`last-value-cache-disable-from-history` *(Enterprise only)*

Parameter Details

Default	FALSE
Allowed Values	FALSE, TRUE
Category	Caching (Enterprise only)

Detailed Explanation:

Controls whether the last value cache is populated from historical Parquet files. When FALSE, the system reads historical files during startup to determine the last known value for each time series.

Recommendation: FALSE (default) for IoT/monitoring dashboards. TRUE for analytics-only workloads or very large series counts (millions).

Category 7: Table Index Cache

`table-index-cache-concurrency-limit`

Parameter Details

Default	20
Allowed Values	Integer: 1 – 100
Category	Table Index Cache

Detailed Explanation:

Limits the number of concurrent operations that can access the table index cache simultaneously. Acts as a concurrency semaphore to prevent cache contention under high query load.

Recommendations by Instance Size:

Instance Size Recommendations

Instance Type	vCPUs	Recommended Value
db.influx.medium	1	**8–16**

Instance Type	vCPUs	Recommended Value
db.influx.large	2	**16–20**
db.influx.xlarge	4	**20**
db.influx.2xlarge	8	**20–24**
db.influx.4xlarge	16	**24–32**
db.influx.8xlarge	32	**32–48**
db.influx.12xlarge+	48+	**48–100**

`table-index-cache-max-entries`

Parameter Details

Default	1000
Allowed Values	Integer: 0+ (0 disables the limit)
Category	Table Index Cache

Detailed Explanation:

Sets the maximum number of table index entries that can be held in the cache. Each entry corresponds to one table's index structure. If your database has more tables than this limit, the least-recently-used table indexes are evicted.

Recommendation: Keep at 1000 (default). If your database has more than 1000 tables, increase to match your table count.

Category 8: HTTP and Network Configuration

`max-http-request-size`

Parameter Details

Default	10,485,760 (10 MiB)

Allowed Values	Long: 1,024 – 16,777,216 (1 KiB – 16 MiB)
Category	Network

Detailed Explanation:

Sets the maximum allowed size for any single incoming HTTP request body. Applies to write, query, and API endpoints. Requests exceeding this size are rejected with HTTP 413.

Recommendation: 5 MiB for db.influx.medium. 10 MiB (default) for db.influx.large through db.influx.xlarge. 10–16 MiB for db.influx.2xlarge and above if clients send large batches.

Category 9: Logging and Observability

`log-filter`

Parameter Details

Default	*(empty)*
Allowed Values	String, max 1,024 characters
Category	Logging

Detailed Explanation:

Configures log filtering using the Rust `tracing` subscriber filter syntax. Examples: `info`, `warn`, `influxdb3_write=debug`, `influxdb3_write=debug,influxdb3_server=info`.

Recommendation: Leave empty (default) for production. Use component-specific filters (e.g., `influxdb3_write=debug`) for targeted troubleshooting. Never leave debug/trace enabled in production long-term.

Category 10: Catalog and Replication

Catalog and replication parameters are available in the **Enterprise edition only**. These parameters control multi-node cluster synchronization.

`catalog-sync-interval`

Parameter Details

Default	1 second
Allowed Values	Duration
Category	Catalog Synchronization

Detailed Explanation:

In Enterprise clusters with multiple nodes, the catalog (metadata about databases, tables, columns, retention policies) must be kept in sync. This parameter controls how frequently each node checks for catalog updates.

Recommendation: 1 second (default) for frequent schema changes. 5–10 seconds for stable schemas. 2–5 seconds for high node counts.

`replication-interval`

Parameter Details

Default	250 ms
Allowed Values	Duration
Category	Replication

Detailed Explanation:

Controls how frequently data written to one node is replicated to other nodes in the Enterprise cluster. This determines the replication lag.

Recommendation: 250ms – 1 second for real-time alerting. 1 second for dashboard monitoring. 5–10 seconds for batch analytics. 5–30 seconds for cost-sensitive bandwidth.

Core and Enterprise versions in Timestream for InfluxDB 3

Amazon Timestream for InfluxDB 3 is available in two versions: Core and Enterprise. Both versions are built to collect, process, transform, and store event and time-series data, and are ideal for

use cases that require real-time ingest and fast query response times to build user interfaces, monitoring, and automation solutions.

Common use cases

Both Core and Enterprise versions are optimized for scenarios where near real-time data monitoring is essential and queries need to return quickly to support user experiences such as dashboards and interactive user interfaces. Common use cases include:

- Monitoring sensor data
- Server monitoring
- Application performance monitoring
- Network monitoring
- Financial market and trading analytics
- Behavioral analytics

Core version

InfluxDB 3 Core is based on the InfluxDB 3 open source release and provides a cost-effective solution for real-time time-series data processing.

Core feature highlights:

- Diskless architecture with Amazon S3 object storage support.
- Fast query response times (under 10ms for last-value queries, or 30ms for distinct metadata queries).
- Embedded Python VM for plugins and triggers .
- Parquet file persistence for efficient storage and querying.
- Full compatibility with InfluxDB 1.x and 2.x write APIs.
- Limited to single-node deployments.

Limitations:

- Lacks compaction capabilities, which impacts performance over time.
- Best suited for workloads focused on recent data (typically a few days old).
- Not recommended for use cases involving long-term storage and analysis.

Enterprise version

InfluxDB 3 Enterprise builds upon Core with additional features designed for demanding production workloads, high availability, and long-term data retention.

Enterprise adds the following features to Core:

- **Historical query capability and single series indexing** for optimized long-term data analysis.
- **High availability** through multi-node cluster configurations.
- **Read replicas** for scaling read-heavy workloads.
- **Compaction capabilities** essential for maintaining performance over time.
- Support for **multi-node deployments** (up to 15 nodes with configurable writer/reader and reader-only node distribution).

> **Tip**
>
> If you have outgrown your Core deployment, you can upgrade to Enterprise edition. See the section called “Upgrade Core to Enterprise”.

Choosing between Core and Enterprise

Choose Core when:

- Your workload focuses on near real-time monitoring of recent data.
- Single-node deployment meets your availability needs.
- Cost optimization is a primary concern.
- You don't require long-term data retention.
- Doing development, testing, or proof-of-concept work to explore engine capabilities.

Choose Enterprise when:

- You need to store and analyze historical data beyond a few days.
- High availability and read scaling are requirements.
- You need compaction to maintain long-term performance.
- Production workloads demand enhanced reliability and scalability.

- You require multi-node deployments for better performance distribution.

Both versions leverage Amazon Timestream for InfluxDB's managed service benefits, including automated backups, software updates, and integration with AWS services, while providing the flexibility to choose the feature set that best matches your workload requirements.

Deployment models in Timestream for InfluxDB 3

Single-node deployment

In a single-node deployment (available for both Core and Enterprise versions), a single instance handles all database operations. This configuration features:

- One node performing multiple critical roles:
  - Writer role: Handles all write operations to the database.
  - Reader role: Processes read queries and data retrieval operations.
  - Compactor role: Optimizes storage by compacting data files (Enterprise version only).
- Core version: Single node operates with writer and reader roles only.
- Enterprise version: Single node operates with writer, reader, and compactor roles for complete functionality.
- Simplified management with all operations consolidated on a single instance.
- Lower cost compared to multi-node deployments.
- Suitable for development, testing, or smaller production workloads.

Single-node deployments are ideal for getting started with Timestream for InfluxDB 3 or for workloads with moderate performance requirements.

Multi-node deployment

Multi-node deployments (available only for Enterprise edition) distribute workloads across multiple instances for improved performance, scalability, and availability. Enterprise clusters support up to 15 nodes with the following configuration:

- Flexible node allocation based on workload requirements:
  - 1-4 writer/reader nodes (ingestQueryInstances) that handle both write and read operations.
  - 0-13 reader-only nodes (queryOnlyInstances) dedicated to processing read queries.

- 1 dedicated compactor node for optimizing storage (required for clusters with 3+ nodes).
- Horizontal scaling capabilities to accommodate growing workloads.
- Enhanced fault tolerance with workload distribution across multiple nodes in different Availability Zones.
- Improved query performance.
- Better handling of concurrent operations.

Multi-node deployments are recommended for production environments with high throughput requirements, large data volumes, or the need for enhanced availability.

Endpoints and connectivity for Timestream for InfluxDB 3

Amazon Timestream for InfluxDB 3 provides multiple endpoints to connect to your cluster, allowing for flexible access patterns based on your application needs.

Endpoint types

Cluster endpoint

The cluster endpoint (or primary endpoint) provides access to the writer nodes in your cluster. This endpoint:

- Supports both read and write operations.
- Automatically routes traffic to available writer nodes.
- Is the only endpoint that can perform write operations.
- Should be used for administrative operations and data ingestion.

Example format: `clusterid-wmyjrrjko.timestream-influxdb-alpha.us-west-2.on.aws`

Reader endpoint

The reader endpoint connects to the reader nodes in your cluster. This endpoint:

- Supports read-only operations (queries).
- Automatically distributes read traffic across all available reader nodes.
- Helps offload query workload from writer nodes.
- Is ideal for reporting and dashboard applications.

Example format: `clusterid-wmyjrrjko-ro.timestream-influxdb-alpha.us-west-2.on.aws`

Node-specific endpoints

In addition to the cluster endpoints, you can connect directly to specific nodes within your cluster:

- Provides direct access to individual nodes for diagnostics or specific workloads.
- Allows for fine-grained control over connection routing.
- Useful for troubleshooting or when you need to isolate specific operations.
- **Important**: During any issue or failover scenario, node-specific endpoints will be temporarily unavailable while the node recovers or is replaced.

Example format: `nodeid-wmyjrrjko.timestream-influxdb-alpha.us-west-2.on.aws`

Traffic distribution

When using cluster-level endpoints (writer/reader and reader endpoints), the traffic distribution system automatically:

- Distributes traffic to active nodes based on:
  - Node health and availability.
  - Current workload on each node.
  - Node role (writer/reader).
- **Makes specific node unavailability transparent to clients** by automatically routing traffic away from unhealthy or unavailable nodes.
- Provides seamless failover without requiring client-side configuration changes.
- Ensures continuous service availability even during maintenance or unexpected node failures.

This automatic traffic distribution provides:

- **High availability**: Client connections remain uninterrupted even when individual nodes become unavailable.
- **Load balancing**: Traffic is distributed across available nodes for optimal performance.
- **Fault tolerance**: Automatic failover ensures minimal impact during node failures.
- **Simplified connection management**: Applications don't need to handle node-level failures.

Connection best practices

For optimal performance and reliability:

- **Use cluster-level endpoints for production workloads**: The writer/reader and reader endpoints provide automatic failover and transparent handling of node unavailability.
- Use the writer/reader endpoint for write operations and administrative tasks.
- Use the reader endpoint for read-heavy applications like dashboards and reports.
- **Avoid node-specific endpoints for critical applications**: These endpoints will be unavailable during failover scenarios and don't provide automatic traffic redistribution.
- Reserve node-specific endpoints for diagnostics, troubleshooting, or when you need to isolate specific operations.
- Implement connection pooling in your applications to efficiently manage connections.
- Configure appropriate timeouts and retry logic in your client applications.

By leveraging the cluster-level endpoints and their automatic traffic distribution capabilities, you can ensure your applications maintain continuous connectivity and optimal performance even during node-level failures or maintenance events.

Managing DNS resolution for cluster endpoints

Timestream for InfluxDB for InfluxDB 3 multi-node clusters use DNS-based traffic distribution to balance connections across nodes. When a failover occurs or nodes are replaced, DNS records automatically update to point to available instances. To ensure your application can discover these changes and maintain optimal traffic distribution, proper DNS resolution configuration is essential.

Understanding DNS caching

Many programming environments cache DNS lookups to improve performance. However, this caching can prevent your application from discovering updated IP addresses after failovers or from distributing connections across multiple cluster nodes. The impact varies by language and runtime:

- **Java/JVM-based applications** — The JVM caches DNS lookups for a configurable time-to-live (TTL). Some configurations cache indefinitely until the JVM restarts.
- **Other languages** — Python, Go, Node.js, and other runtimes typically rely on operating system DNS resolution and may not exhibit the same caching behavior.

Solution 1: Configure JVM DNS TTL (Java applications)

For Java applications connecting to Timestream for InfluxDB for InfluxDB 3 cluster endpoints, set the JVM DNS cache TTL to **zero**. This ensures each connection triggers a fresh DNS lookup, enabling proper traffic distribution and immediate failover detection.

Check your current TTL setting:

```
String ttl = java.security.Security.getProperty("networkaddress.cache.ttl");
```

Configure the TTL using one of these methods:

- **Global configuration** — Set `networkaddress.cache.ttl` in `$JAVA_HOME/jre/lib/security/java.security`:

  ```
  networkaddress.cache.ttl=0
  ```

- **Application-specific configuration** — Set the property in your application initialization code before establishing network connections:

  ```
  java.security.Security.setProperty("networkaddress.cache.ttl", "0");
  ```

- **JVM argument** — Pass as a system property when starting your application:

  ```
  java -Dsun.net.inetaddr.ttl=0 -jar your-application.jar
  ```

> **Note**
>
> For non-cluster endpoints or general AWS resources, a TTL of 60 seconds is typically sufficient. The zero TTL recommendation is specific to InfluxDB 3 cluster endpoints where traffic distribution and rapid failover detection are critical.

Solution 2: Direct resolution via authoritative nameservers

If TTL configuration is insufficient — for example, due to intermediate OS-level caching, or if your programming language doesn't exhibit DNS caching issues — you can force fresh DNS resolution by querying authoritative nameservers directly. Use this approach only when standard TTL configuration doesn't resolve the issue.

Direct resolution workflow

1. Identify the authoritative nameserver for your cluster endpoint:

   ```
   dig <cluster-endpoint> NS
   ```

   In the `AUTHORITY SECTION` of the output, note the nameserver listed after `IN SOA`. For example: `ns-1458.awsdns-54.org`.

2. Query the authoritative nameserver directly to bypass caches:

   ```
   dig @ns-1458.awsdns-54.org <cluster-endpoint>
   ```

   This returns the current IP addresses for the endpoint, reflecting the actual DNS-based traffic distribution.

3. Use the resolved IP address(es) for your application connections. Repeat this resolution periodically or when connection errors occur to refresh addresses.

Example:

```
# Find authoritative nameserver
dig my-cluster.timestream-influxdb.us-east-1.amazonaws.com NS

# Resolve using authoritative nameserver
dig @ns-1458.awsdns-54.org my-cluster.timestream-influxdb.us-east-1.amazonaws.com

# Use returned IP addresses in your application
```

Handling node IP changes

> **Important**
>
> Node IP addresses are **not static**. They change during failovers, node restarts, maintenance operations, and cluster scaling. Never cache resolved IP addresses indefinitely.

Implement these practices to handle IP changes:

- **Periodic re-resolution** — Re-resolve endpoints every 30-60 seconds using a background task. Compare new IP addresses against cached values and update your connection pool accordingly.
- **Error-driven re-resolution** — When a connection fails (timeout, connection refused, reset), immediately re-resolve the endpoint and retry with updated IP addresses.
- **Graceful connection draining** — When an IP address is no longer in the DNS record set, allow in-flight requests to complete but stop creating new connections to that IP.
- **New connection creation** — After re-resolution, create new connections using updated IP addresses. Existing connections pinned to old IPs won't benefit from re-resolution.

Best practices

- **Start with TTL configuration** — For Java applications, always try setting `networkaddress.cache.ttl=0` first. This is the simplest and most effective solution.
- **Use direct resolution sparingly** — Only use authoritative nameserver queries when TTL configuration is insufficient or when dealing with persistent intermediate caching.
- **Automate nameserver discovery** — Don't hardcode nameserver addresses. Discover them dynamically as nameserver assignments can change.
- **Applies to cluster endpoints only** — These techniques are for cluster-level endpoints (write and read) that use DNS-based distribution. Node-specific endpoints that target individual nodes directly don't require this configuration.
- **Test your implementation** — Verify that your application correctly distributes connections across multiple nodes and recovers from simulated node failures.

Manage InfluxDB 3 Clusters in Timestream

Amazon Timestream for InfluxDB 3 provides comprehensive management capabilities for your clusters through the AWS Management Console, AWS CLI, and API. This section covers the essential operations for creating, updating, and deleting InfluxDB 3 clusters.

Topics

- Create a cluster in Timestream for InfluxDB 3
- Update a cluster in Timestream for InfluxDB 3
- Upgrade from Core to Enterprise edition
- Delete a cluster in Timestream for InfluxDB 3

- Reboot a cluster in Timestream for InfluxDB 3
- Scaling a cluster

Create a cluster in Timestream for InfluxDB 3

Creating a new InfluxDB 3 cluster in Amazon Timestream involves selecting the appropriate version, deployment model, and configuration settings to meet your workload requirements.

Using the AWS Management Console

1. Sign in to the AWS Management Console and open the Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**.
3. Choose **Create InfluxDB database**.
4. For **Engine Version**, choose **InfluxDB 3** as your engine version
5. For **Edition**:
   - Choose **Core** for development, testing, or near real-time monitoring of recent data.
   - Choose **Enterprise** for production workloads requiring high availability, long-term storage, and compaction.
6. Configure Deployment (based on **Edition** selected).
   - For Core Edition:
     - Automatically configured as single-node deployment
     - No additional node configuration required
   - For Enterprise Edition:
     - Choose deployment configuration:
       - **Single-node**: One node handling writer, reader, and compactor roles
       - **3-node cluster**: 2 writer/reader nodes + 1 dedicated compactor node
       - **Multi-node cluster (up to 15 nodes)**: Configure via custom parameter group with:
         - 1-4 writer/reader nodes (ingestQueryInstances)
         - 0-13 reader-only nodes (queryOnlyInstances)
         - 1 dedicated compactor (required for clusters with 3+ nodes)

> **Note**
>
> To create a multi-node cluster with more than 3 nodes, you must first create a custom parameter group with your desired node configuration, then create the cluster using that parameter group.

7. Configure cluster settings
   - **DB cluster identifier**: Enter a unique name for your cluster.
   - **DB instance class**: Select from `db.influx` instance classes (applies to all nodes).
   - **Parameter group**:
     - Select a service-defined parameter group:
       - InfluxDBv3Core (for Core single-node)
       - InfluxDBv3Enterprise (for Enterprise 3-node)
       - InfluxDBv3Enterprise1Node (for Enterprise single-node)
     - Or create/select a custom parameter group for multi-node configurations
8. Configure network settings
   - **Virtual Private Cloud (VPC)**: Select your VPC.
   - **DB subnet group**: Choose subnets across availability zones.
   - **VPC security groups**: Select security groups for network access control.
   - Public accessibility:
     - **Yes**: Cluster accessible from internet (with proper security group rules).
     - **No**: Cluster only accessible within VPC.

> **Note**
>
> Shared VPCs are not currently supported for Timestream for InfluxDB 3.

> **Note**
>
> For multi-node clusters, nodes are automatically distributed across multiple Availability Zones for high availability.

9. (Optional) Configure additional settings
   - **Tags**: Add metadata tags for organization and billing.
10. Review and create
   - Review all configuration settings.
   - Choose **Create InfluxDB database**.

The cluster creation process typically takes 10-20 minutes. The console will display the status as "Creating" and change to "Available" when ready.

Using the AWS CLI

Create an InfluxDB 3 Core cluster:

```
aws timestream-influxdb create-db-cluster \
  --region us-east-1 \
  --name "my-influxdb3-core-cluster" \
  --db-parameter-group-identifier "InfluxDBv3Core" \
  --db-instance-type db.influx.large \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --publicly-accessible false
```

Create an InfluxDB 3 Enterprise cluster (3-node):

```
aws timestream-influxdb create-db-cluster \
  --region us-east-1 \
  --name "my-influxdb3-enterprise-cluster" \
  --db-parameter-group-identifier "InfluxDBv3Enterprise" \
  --db-instance-type db.influx.xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --publicly-accessible false \
  --log-delivery-configuration '{
    "s3Configuration": {
      "bucketName": "my-influxdb-logs",
      "enabled": true
    }
  }'
```

Create an InfluxDB 3 Enterprise cluster (single-node):

```
aws timestream-influxdb create-db-cluster \
  --region us-east-1 \
  --name "my-influxdb3-enterprise-single" \
  --db-parameter-group-identifier "InfluxDBv3Enterprise1Node" \
  --db-instance-type db.influx.large \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --publicly-accessible false
```

Create a cluster with a custom parameter group:

```
# First, create a custom parameter group
aws timestream-influxdb create-db-parameter-group \
  --name "custom-enterprise-pg" \
  --engine-type "InfluxDBv3Enterprise" \
  --parameters '{
    "InfluxDBv3Enterprise": {
      "ingestQueryInstances": 2,
      "queryOnlyInstances": 0,
      "dedicatedCompactor": true,
      "bufferMemLimitMb": 32768,
      "dataFusionNumThreads": 8
    }
  }'

# Then create the cluster using the parameter group identifier
aws timestream-influxdb create-db-cluster \
  --region us-east-1 \
  --name "my-custom-enterprise-cluster" \
  --db-parameter-group-identifier "<parameter-group-identifier>" \
  --db-instance-type db.influx.2xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc
```

Note

The `--db-parameter-group-identifier` requires the parameter group identifier (not the name). Set `dataFusionNumThreads` to match the number of vCPUs for your instance type (for example, 8 for `db.influx.2xlarge`). For a complete guide on parameter

configuration, see the section called "Parameter Groups for DB Clusters in Amazon Timestream".

Create a multi-node cluster (up to 15 nodes) with a custom parameter group:

```
# First, create a custom parameter group with your desired node configuration
aws timestream-influxdb create-db-parameter-group \
  --name "custom-multinode-pg" \
  --engine-type "InfluxDBv3Enterprise" \
  --parameters '{
    "InfluxDBv3Enterprise": {
      "ingestQueryInstances": 4,
      "queryOnlyInstances": 10,
      "dedicatedCompactor": true,
      "bufferMemLimitMb": 32768,
      "dataFusionNumThreads": 8
    }
  }'

# Then create the cluster using the parameter group identifier
aws timestream-influxdb create-db-cluster \
  --region us-east-1 \
  --name "my-multinode-enterprise-cluster" \
  --db-parameter-group-identifier "<parameter-group-identifier>" \
  --db-instance-type db.influxIOIncluded.2xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc
```

Post-creation steps

After your cluster is created and available:

1. **Retrieve endpoints:**

   ```
   aws timestream-influxdb get-db-cluster \
     --identifier "my-influxdb3-cluster-id"
   ```

   For multi-node clusters, you will receive:

   - **Cluster read/write endpoint**: Distributes traffic to writer/reader nodes

- **Cluster read-only endpoint**: Distributes traffic to all nodes capable of read operations. This endpoint is only created when at least one reader-only node is present in the cluster. If your cluster has no reader-only nodes, the read-only endpoint will not be available.
- **Node endpoints**: Direct access to specific nodes for workload isolation

2. **Obtain your operator token:**
   - Your operator token is stored in a secret in your AWS Secrets Manager account. The secret has the same name as your newly created cluster identifier.
3. (Optional) **Access InfluxDB explorer:**
   - Download the InfluxDB Explorer from https://docs.influxdata.com/influxdb3/explorer/.
   - For public clusters: Run the Explorer from any location with internet access.
   - For private clusters: Run the Explorer from within the same VPC (using an EC2 instance, bastion host, or through VPN connection).
4. **Verify connectivity:**

```
influxdb3 query \
  --host "your-cluster-endpoint:8086" \
  --database "my-database" \
  --token "my-token" \
  "SHOW TABLES"
```

Please refer to InfluxDB 3 documentation to find information on writing data, executing queriers, or Administer your influxDB 3 Database.

Important considerations:

- **Edition Selection**: Core is suitable for development and short-term data; Enterprise is required for production workloads with long-term storage needs.
- **Node Configuration**: All nodes in a cluster share the same instance class and network configuration.
- **Parameter Groups**: Immutable once created; changes require creating a new parameter group.
- **Scaling**: Initial release supports 1-node and 3-node configurations; future updates will enable additional scaling options. At initial release you won't be able to change your node count after creation.
- **High Availability**: 3-node Enterprise configurations provide better fault tolerance and performance distribution.

- **Compaction**: Enterprise edition's compaction capability is essential for maintaining performance over time.

Update a cluster in Timestream for InfluxDB 3

After your cluster is created, you can modify certain configuration parameters to adapt to changing workload requirements in Amazon Timestream. Note that some modifications may cause temporary downtime during the update process.

Update a cluster using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**.
3. Select the cluster you want to modify.
4. Choose **Modify**.
5. Update available settings:
   - **Instance class**: Scale compute resources up or down (applies to all nodes in the cluster and requires cluster restart).
   - **Parameter group**: Associate a different parameter group to change engine configuration or cluster shape (requires cluster restart).
   - **Log delivery configuration**: Enable or modify CloudWatch Logs export.
   - **Edition**: Upgrade from Core to Enterprise (one-way). For details, see the section called "Upgrade Core to Enterprise".
6. Choose **Continue** and review your changes.
7. Select when to apply changes:
   - **Apply immediately**: Changes are applied right away (may cause brief downtime).
8. Choose **Modify cluster**.

Update a cluster using the AWS CLI

Update instance type:

```
aws timestream-influxdb update-db-cluster \
```

```
  --region us-east-1 \
  --identifier "my-influxdb3-cluster" \
  --db-instance-type db.influxIOIncluded.2xlarge \
  --apply-immediately
```

Update parameter group (including cluster scaling):

To modify your cluster configuration—including scaling the number of nodes—create a new parameter group with your desired configuration and apply it to your cluster:

```
# Step 1: Create a new parameter group with desired node configuration
aws timestream-influxdb create-db-parameter-group \
  --name "scaled-enterprise-pg" \
  --parameters '{
    "InfluxDBv3Enterprise": {
      "ingestQueryInstances": 4,
      "queryOnlyInstances": 10,
      "dedicatedCompactor": true,
      "dataFusionNumThreads": 8,
      "bufferMemLimitMb": 32768
    }
  }'

# Step 2: Apply the new parameter group to your cluster using the parameter group
 identifier
aws timestream-influxdb update-db-cluster \
  --region us-east-1 \
  --identifier "my-influxdb3-cluster" \
  --db-parameter-group-identifier "<parameter-group-identifier>" \
  --apply-immediately
```

Scaling your cluster (Enterprise edition)

For Enterprise edition clusters, you can scale your cluster size by applying a new parameter group with your desired node configuration. For detailed guidance, see the section called “Scaling a cluster”.

- **Scale up**: Add nodes by increasing ingestQueryInstances (up to 4) or queryOnlyInstances (up to 13)
- **Scale down**: Remove nodes by decreasing these values
- **Change node modes**: Adjust the ratio of writer/reader nodes to reader-only nodes

Important: Clusters with 3 or more nodes require a dedicated compactor (dedicatedCompactor: true). When scaling, nodes are automatically distributed across multiple Availability Zones for high availability.

Considerations when updating a cluster

Modifiable parameters

The following parameters can be modified after cluster creation:

Parameter	Description	Impact
Instance class	Scale compute/memory resources up or down	Requires rolling restart of nodes
Parameter group	Change database engine configuration or cluster shape (node count and modes)	Requires cluster restart
Log delivery	Configure CloudWatch Logs export	Applied immediately
Edition	Upgrade from Core to Enterprise (one-way). See the section called "Upgrade Core to Enterprise".	Requires cluster restart

Non-modifiable parameters

The following parameters **cannot** be modified after cluster creation:

- **VPC and Subnets**: Network location is fixed.
- **Security Groups**: Network access controls cannot be changed.
- **Public Accessibility**: Internet accessibility setting is permanent.
- **DB Cluster Identifier**: Cluster name cannot be changed.

Downtime impact:

- Instance class changes: Rolling restart (minimal downtime with multi-node clusters).

- Parameter group changes (including cluster scaling): Cluster restart required. For multi-node clusters using cluster endpoints, traffic is automatically redistributed to available nodes during the update process.
- Log configuration changes: No downtime.

Best practices:

- **Plan network configuration carefully** before cluster creation as it cannot be changed.
- **Test changes** in non-production environments first.
- **Monitor metrics** during and after updates to verify performance.
- **Use cluster endpoints** to ensure transparent failover during rolling updates and scaling operations.
- **Avoid node-specific endpoints** during update operations, as individual nodes may be restarted or replaced.
- **When scaling**, consider your workload patterns to determine the optimal ratio of writer/reader nodes to reader-only nodes.

Validation:

- The system validates all changes before applying them.
- Invalid configurations are rejected with descriptive error messages.
- Some parameter combinations may not be compatible (e.g., clusters with 3+ nodes require a dedicated compactor).

Rollback:

- Keep previous parameter group configurations for easy rollback.
- Document current settings before making changes.
- For critical changes, create cluster snapshots before updating.
- To rollback a scaling operation, create a parameter group with your previous node configuration and apply it to the cluster.

Monitor update progress

Track the update status through:

1. **Console**: The cluster status shows "Modifying" during updates.
2. **CLI**: Query cluster status:

   ```
   aws timestream-influxdb get-db-cluster \
     --identifier "my-influxdb3-cluster"
   ```

   The possible status for a cluster are:

   - `CREATING`
   - `UPDATING`
   - `DELETING`
   - `AVAILABLE`
   - `FAILED`
   - `DELETED`
   - `UPDATING_INSTANCE_TYPE`
   - `PARTIALLY_AVAILABLE` (Applicable only for multi-node clusters when some nodes are still being created or updated)
3. **CloudWatch Metrics**: Monitor performance metrics during updates.

Limitations

Current limitations for cluster updates:

- Cannot modify VPC, subnets, security groups, or public accessibility after creation.
- Upgrading from Core to Enterprise is supported (one-way), but you cannot downgrade from Enterprise to Core. See the section called “Upgrade Core to Enterprise”.
- Cannot rename the cluster (DB cluster identifier is immutable).
- All nodes must use the same instance class (except for the compactor).
- Parameter groups are immutable (must create new ones for changes).
- Cluster scaling operations (adding/removing nodes) require creating and applying a new parameter group.

By understanding these update capabilities and limitations, you can effectively manage and scale your InfluxDB 3 clusters to meet evolving workload requirements while planning appropriately for settings that must be configured at cluster creation time.

Upgrade from Core to Enterprise edition

You can upgrade an existing InfluxDB 3 Core cluster to Enterprise edition to gain access to features such as multi-node deployments, long-term data retention, and dedicated compaction.

> **Important**
>
> Upgrading from Core to Enterprise is a **one-way operation**. Once a cluster is upgraded to Enterprise, it cannot be reverted to Core edition.

Prerequisites and requirements

- **First-time Enterprise activation**: If this is the first time you are activating an Enterprise license on your AWS account, the upgrade must be performed through the AWS Management Console. This one-time console activation enables Enterprise capabilities for your account.
- **Subsequent operations**: After the initial console activation, your account is enabled for Enterprise. You can then upgrade additional Core clusters to Enterprise or deploy new Enterprise clusters using the AWS CLI, API, or the AWS Management Console.

Upgrade using the AWS Management Console

1. Sign in to the AWS Management Console and open the Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**.
3. Select the Core cluster you want to upgrade.
4. Choose **Modify**.
5. For **Edition**, select **Enterprise**.
6. Review the changes and choose **Modify cluster**.

What happens during the upgrade

When you upgrade a Core cluster to Enterprise:

1. **Cluster restart**: The cluster restarts to apply the Enterprise engine configuration.
2. **Node configuration**: Depending on your cluster node count, the cluster will be configured as either:
   - A single-node Enterprise cluster (all-in-one: writer, reader, and compactor on one node)
   - A multi-node Enterprise cluster with a dedicated compactor (for clusters with 3 or more nodes)
3. **Data compaction**: Your existing data will be gradually compacted in the background by the Enterprise compaction engine. The cluster remains usable during this process.

> **Note**
>
> The time required for data compaction to catch up depends on the volume of existing data, the cluster size, instance size, and the available CPU and memory headroom relative to your current workload.

> **Important**
>
> **Capacity considerations for compaction after upgrade**
>
> - **Single-node clusters**: When upgrading a Core single-node cluster to Enterprise, the compactor runs on the same node as the writer and reader. Because Core does not include a compactor, all existing data must be compacted after the upgrade. This requires additional CPU and memory capacity beyond what your current workload uses. If the node does not have sufficient headroom, compaction may compete with your read and write workloads, potentially impacting performance. Consider scaling up to a larger instance type (for example, from `db.influx.xlarge` to `db.influx.2xlarge`) before or immediately after the upgrade to give the compactor the resources it needs. You can scale back down after compaction catches up.
> - **Multi-node (3-node) clusters**: When upgrading to a 3-node Enterprise cluster, a dedicated compactor node is provisioned. However, the full benefits of Enterprise—such as optimized query performance from compacted data—will not be realized until the compactor finishes processing all pre-existing data. The time this takes depends on the volume of data already in the database, the current running workload, and the instance size and configuration of the compactor node.

Tuning compaction performance: To help the compactor process existing data faster, you can adjust the following Enterprise-only parameters in your parameter group:

- `compaction-max-num-files-per-plan` – Increase this value to allow the compactor to process more files per compaction cycle. For example, increase from the default of 500 to 1000–5000 on larger instances (db.influx.4xlarge and above). See the section called "`compaction-max-num-files-per-plan`" for recommended values by instance size.
- `compaction-check-interval` – Reduce from the default of 10 seconds to 5 seconds on db.influx.4xlarge and above to make the compactor evaluate work more frequently.

For single-node clusters, scaling up the instance type is the most effective way to give the compactor more capacity, since all roles share the same node's resources. For multi-node clusters, the dedicated compactor node uses the same instance class as the other nodes, so scaling the cluster's instance type also increases compactor capacity. For full details on compaction parameters, see the section called "Category 4: Compaction".

Considerations

- This is a one-way upgrade. You **cannot** downgrade from Enterprise to Core.
- Plan the upgrade during a period of lower activity to allow the compaction process sufficient CPU and memory headroom to catch up.
- **Single-node clusters**: Consider temporarily scaling up to a larger instance type before upgrading to ensure the compactor has enough CPU and memory to compact existing data without impacting your workload.
- **Multi-node clusters**: Expect a transition period after upgrading where the compactor is processing all pre-existing data. Full Enterprise performance benefits are realized after compaction completes. The duration depends on data volume, workload intensity, and instance size.
- You can speed up post-upgrade compaction by increasing `compaction-max-num-files-per-plan` and reducing `compaction-check-interval` in your parameter group. See the section called "Category 4: Compaction" for details.
- After upgrading, you can take advantage of Enterprise features such as the section called "Scaling a cluster" to scale your cluster horizontally.

Delete a cluster in Timestream for InfluxDB 3

When you no longer need a cluster in Amazon Timestream, you can delete it to avoid incurring additional charges.

> **Important**
>
> Deletion permanently removes all data, so ensure you have exported any critical data before proceeding.

Delete a cluster using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**.
3. Select the cluster you want to delete.
4. Choose **Delete** from the Actions menu.
5. In the confirmation dialog box, confirm the deletion and acknowledge that all data will be permanently deleted.
6. Choose **Delete**.

The deletion process typically takes five to ten minutes. The cluster status will show "Deleting" and will be removed from the list once deletion is complete.

Delete a cluster using the AWS CLI

To delete a cluster using the AWS CLI:

```
aws timestream-influxdb delete-db-cluster \
  --region us-east-1 \
  --identifier "my-influxdb3-cluster"
```

Considerations when deleting a cluster

Data loss warning:

- **Permanent deletion**: All data is permanently removed with no recovery option.

- **Manual backup required**: Export critical data using InfluxDB tools before deletion.
- **No built-in snapshot**: The service does not currently provide snapshot capabilities.

  No built-in customer-managed snapshot: The service does not currently provide customer-managed snapshot capabilities. The service level snapshot is built with disaster recovery in mind and is kept for 24 hours.

What gets deleted:

- All cluster nodes (writer, reader, and compactor nodes).
- All data stored in InfluxDB Object Storage (Amazon S3).
- Cluster endpoints and DNS entries.

What is retained:

- Database logs exported to Amazon S3 (if configured).
- Parameter groups (not automatically deleted).
- VPC security groups (not automatically deleted).

Dependencies to Check:

- **Active connections**: Ensure no applications are actively using the cluster.
- **Scheduled jobs**: Disable any scheduled data ingestion or query jobs.
- **Monitoring**: Update or remove CloudWatch alarms associated with the cluster.
- **Documentation**: Update any documentation referencing the cluster endpoints.

Pre-deletion checklist:

Critical: Before deleting your InfluxDB 3 cluster, manually export all important data:

1. **Export critical data:** Use the following query to export data to JSON format. For more information, see Output format for the `influxdb3 query` command in the *Influx DB 3 documentation*.

```
influxdb3 query \
  --token AUTH_TOKEN \
```

```
  --database DATABASE_NAME \
  --format json \
  "SELECT * FROM home WHERE time >= '2025-06-26T08:00:00Z' LIMIT 5
```

2. **Document cluster configuration:**

```
aws timestream-influxdb describe-db-cluster \
  --identifier "my-influxdb3-cluster" > cluster-config-backup.json
```

3. **Save parameter group configuration:**

```
aws timestream-influxdb describe-db-parameter-group \
  --identifier "your-parameter-group-id" > parameter-group-backup.json
```

4. **Check for active connections**
   - Review CloudWatch metrics for active connections.
   - Verify no critical applications are using the cluster.

Billing implications:

- **Immediate cessation**: Compute charges stop once deletion begins.
- **Storage charges**: Object storage charges stop after data is deleted.
- **Minimum billing**: No minimum billing period applies for deletion.

Recovery options after deletion:

- **No automatic recovery**: Deleted clusters and data cannot be recovered.
- **Manual restoration only**: You must recreate the cluster and reimport data from manual backups.
- **Same name reuse**: The cluster identifier cannot be reused after deletion is complete.

The following table covers some common deletion issues and solutions:

Issue	Solution
Deletion fails	Check for dependent resources or AWS service issues.
Slow deletion	Large clusters may extend deletion time.

Issue	Solution
Cannot delete	Ensure you have proper IAM permissions.
Need to preserve data	Must export data manually before deletion.

Best Practices for deleting a cluster

- **Always export critical data** before deletion.
- **Test data export and reimport process** before deleting production clusters.
- **Create comprehensive backups** including schema, data, and configuration.
- **Document deletion** for audit and compliance purposes.
- **Coordinate with teams** before deleting shared resources.
- **Verify deletion completion** before removing related resources.

Reboot a cluster in Timestream for InfluxDB 3

You can reboot your cluster in Amazon Timestream in the event of a cluster health issue.

Reboot a cluster using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **InfluxDB Databases**.
3. Select the cluster you want to reboot.
4. Choose **Restart database** from the Actions menu.
5. Choose **Confirm and Restart**.

Rebooting a DB cluster restarts the database engine service in all the instances that are part of the cluster. Rebooting a DB cluster results in a momentary outage, during which the DB cluster status is set to rebooting.

Rebooting a cluster using the AWS CLI

To reboot a cluster using the AWS CLI:

```
aws timestream-influxdb reboot-db-cluster \
  --region us-east-1 \
  --identifier "my-influxdb3-cluster"
```

Scaling a cluster

Overview

Multi-node scaling extends the capabilities of Timestream for InfluxDB 3 Enterprise edition by allowing you to create clusters with up to 15 nodes, enabling horizontal scalability for increased workload demands. This feature provides flexible cluster configuration options to optimize for specific read and write workloads, with nodes distributed across multiple Availability Zones for high availability.

Key capabilities

With multi-node scaling, you can:

- Create clusters with up to 15 nodes (maximum 4 writer nodes, 13 reader nodes, and 1 dedicated compactor)
- Dynamically adjust cluster size by adding or removing nodes as your workload requirements change
- Configure node roles between Writer/Reader mode or Reader-only mode
- Distribute nodes across multiple Availability Zones for improved availability
- Access your cluster through dedicated endpoints for read/write or read-only operations

Cluster configuration

Node types and roles

When configuring your cluster, you can specify:

- **Writer/Reader Nodes**: Process both write and read operations (1-4 nodes)
- **Reader-only Nodes**: Dedicated to processing read queries (0-13 nodes)
- **Dedicated Compactor**: Required for clusters with 3+ nodes to optimize data storage

Cluster endpoints

Timestream for InfluxDB 3 provides two types of endpoints:

- **Cluster Endpoints**: Distribute traffic among available nodes
  - **Read/Write Endpoint**: Routes traffic to all Writer/Reader nodes
  - **Read-only Endpoint**: Routes traffic to all nodes capable of read operations
- **Node Endpoints**: Allow direct access to specific nodes for workload isolation

> **Important**
>
> When using node endpoints, there will be downtime if the node is restarted, patched, upgraded, or fails.

> **Note**
>
> When generating writes or reads using multiple parallel threads from the same client machine, DNS caching behavior can affect traffic distribution across cluster nodes. For guidance on optimizing DNS resolution to ensure even traffic distribution, see Managing DNS resolution for cluster endpoints.

Managing cluster size

To modify your cluster configuration, you must use parameter groups:

1. Create a new parameter group with your desired configuration
2. Apply the parameter group to your cluster

This approach applies to both scaling up (adding nodes) and scaling down (removing nodes).

> **Important**
>
> When creating a new parameter group for scaling, ensure you are using all the recommended parameters for your instance size. Review the the section called "Supported Instance Types and Specifications" page for instance-specific guidance. Before applying

the new parameter group, verify your current running configuration to ensure no settings are accidentally changed when adding or removing nodes. You can check your effective configuration by following the steps in the section called "Creating Parameter Groups with the AWS CLI".

Multi-AZ deployment

When running multi-node clusters, nodes are automatically distributed across different Availability Zones in your selected region. This distribution ensures:

- No downtime in your cluster when using Cluster endpoints
- Improved fault tolerance against AZ-level failures
- Consistent performance across the region

Requirements and limitations

- Clusters with 3+ nodes require a dedicated compactor
- A minimum of 2 Writer/Reader nodes is recommended for high availability
- All nodes in a cluster must use the same instance type
- Node mode changes require creating a new parameter group
- Node endpoints may experience downtime during maintenance operations

Best practices

- Use cluster endpoints instead of node endpoints for production workloads to ensure high availability
- Scale your cluster based on your read/write workload ratio (more reader nodes for read-heavy workloads)
- Monitor node performance to determine optimal scaling needs
- Plan scaling operations during periods of lower activity
- To maximize performance on write-heavy workloads in multi-node deployments, send write operations only to your writer node(s) and route read queries to your reader-only nodes

> **Tip**
>
> For write-intensive workloads, configure more Writer/Reader nodes. For read-intensive workloads, add more Reader-only nodes while maintaining at least 2 Writer/Reader nodes for high availability.

Maintenance windows for Timestream for InfluxDB 3

Amazon Timestream for InfluxDB performs periodic maintenance on your resources. Maintenance most often involves updates to the underlying hardware, operating system, or database engine. These updates are mandatory. If maintenance is not performed during the designated window, it may be applied during a subsequent window.

How maintenance windows work

Every Timestream for InfluxDB resource has a weekly maintenance window during which maintenance is performed.

- If you specify a preferred maintenance window, maintenance is performed during your chosen window.
- If you do not specify a preferred maintenance window, the system automatically assigns a maintenance window for your resource.

You can set your preferred maintenance window when creating a resource or change it later using the update APIs.

> **Important**
>
> If a required maintenance action has been deferred for more than 25 days, the service may apply maintenance outside of your preferred window to ensure the security and reliability of your resource.

Management types

Maintenance windows can be managed in two ways:

Type	Description	Behavior
`SERVICE_MANAGED`	AWS manages maintenance (default)	AWS determines the optimal maintenance window
`CUSTOMER_MANAGED`	Customer manages maintenance	You must specify a `preferredMaintenanceWindow`

Preferred maintenance window

The preferred maintenance window defines the weekly time slot during which maintenance can occur.

Format: `ddd:HH:MM-ddd:HH:MM`

- ddd — Day of the week (`Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`). Case-insensitive.
- `HH:MM` — Time in 24-hour format.

Examples:

Window	Description	Duration
`Sun:02:00-Sun:04:00`	Sunday 2:00 AM to 4:00 AM	2 hours
`Sat:22:00-Sun:02:00`	Saturday 10:00 PM to Sunday 2:00 AM	4 hours
`Wed:03:30-Wed:05:30`	Wednesday 3:30 AM to 5:30 AM	2 hours

Constraints:

Constraint	Value
Minimum window duration	2 hours
Maximum window duration	24 hours
Cross-midnight windows	Supported (for example, `Sat:22:00-Sun:02:00`)

Maintenance begins within the first 30 minutes of your specified window. The exact start time is determined by the system and is consistent week to week for the same resource.

Timezone

You can specify a timezone for your maintenance window using the `timezone` field. When set, window times are interpreted in the specified timezone.

The `timezone` field is required in all maintenance schedule requests.

Format: IANA timezone identifier. Most timezones in the IANA Time Zone Database are supported, including common identifiers such as `America/New_York`, `Europe/London`, `Asia/Tokyo`, `Asia/Kolkata`, and `Pacific/Auckland`.

> **Note**
>
> Timezone abbreviations such as `EST`, `PST`, and `GMT+5` are not supported. Use the full IANA identifier instead.

Daylight Saving Time: The system handles DST transitions automatically. If your window falls within a skipped DST hour (spring forward), maintenance is skipped that week. To avoid DST-related issues, schedule maintenance outside the 1:00 AM to 3:00 AM range, or use UTC.

Supported timezones:

Timezone	Description
`UTC`	Coordinated Universal Time (default)
`America/New_York`	US Eastern
`America/Chicago`	US Central
`America/Denver`	US Mountain
`America/Los_Angeles`	US Pacific
`America/Sao_Paulo`	Brazil
`Europe/London`	UK

Timezone	Description
`Europe/Paris`	Central Europe
`Europe/Berlin`	Germany
`Asia/Tokyo`	Japan
`Asia/Shanghai`	China
`Asia/Singapore`	Singapore
`Asia/Mumbai`	India
`Asia/Dubai`	UAE
`Australia/Sydney`	Australia Eastern
`Pacific/Auckland`	New Zealand

During maintenance

When maintenance is being performed, the resource status changes to `MAINTENANCE`. During this time, the resource may be temporarily unavailable. After completion, the status returns to `AVAILABLE`.

API response fields

Field	Description
`preferredMaintenanceWindow`	The configured maintenance window. Shows system-assigned window if not customer-specified.
`timezone`	The configured timezone.
`nextMaintenanceTime`	Next scheduled maintenance time in UTC (ISO 8601).
`lastMaintenanceTime`	Last completed maintenance time in UTC (ISO 8601). Null if never performed.

> **Note**
>
> `nextMaintenanceTime` and `lastMaintenanceTime` are always returned in UTC, regardless of your configured timezone.

CLI examples

Setting a maintenance window when creating a resource

Service managed (default — no maintenance schedule specified):

```
aws timestream-influxdb create-db-cluster \
  --name "my-influxdb3-cluster" \
  --db-instance-type db.influx.2xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --region us-west-2
```

Customer managed with UTC (default timezone):

```
aws timestream-influxdb create-db-cluster \
  --name "my-influxdb3-cluster" \
  --db-instance-type db.influx.2xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --maintenance-schedule '{
    "preferredMaintenanceWindow": "Sun:02:00-Sun:04:00"
  }' \
  --region us-west-2
```

Customer managed with a specific timezone:

```
aws timestream-influxdb create-db-cluster \
  --name "my-influxdb3-cluster" \
  --db-instance-type db.influx.2xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --maintenance-schedule '{
    "timezone": "America/New_York",
```

```
    "preferredMaintenanceWindow": "Sun:02:00-Sun:04:00"
  }' \
  --region us-west-2
```

Customer managed with an overnight window:

```
aws timestream-influxdb create-db-cluster \
  --name "my-influxdb3-cluster" \
  --db-instance-type db.influx.2xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc \
  --maintenance-schedule '{
    "timezone": "Europe/London",
    "preferredMaintenanceWindow": "Sat:22:00-Sun:02:00"
  }' \
  --region us-west-2
```

Updating a maintenance window

Set a custom maintenance window on an existing resource:

```
aws timestream-influxdb update-db-cluster \
  --identifier "my-influxdb3-cluster" \
  --maintenance-schedule '{
    "timezone": "Asia/Tokyo",
    "preferredMaintenanceWindow": "Wed:03:00-Wed:06:00"
  }' \
  --region us-west-2
```

Revert to service managed:

```
aws timestream-influxdb update-db-cluster \
  --identifier "my-influxdb3-cluster" \
  --maintenance-schedule '{
    "timezone": "UTC",
    "preferredMaintenanceWindow": ""
  }' \
  --region us-west-2
```

Considerations

- Maintenance windows define when maintenance *can* occur, not when it *will* occur. Maintenance is performed as needed, typically no more than once per week.
- AWS requires maintenance to be performed at least once per month for security and reliability patching.
- For multi-node clusters using cluster endpoints, traffic is automatically redistributed to available nodes during maintenance.
- Schedule your maintenance window during periods of lowest traffic to minimize impact on your workload.
- If you use a timezone with DST transitions, avoid scheduling maintenance between 1:00 AM and 3:00 AM to prevent skipped windows during spring forward.

Customer-managed backup and restore

Customer-managed backup and restore lets you create, manage, and restore database backups for your Amazon Timestream for InfluxDB 3 clusters (Core and Enterprise). You can create on-demand backups, configure automated scheduled backups, or enable continuous backups for point-in-time restore—all without opening AWS support tickets.

Overview

The first backup creates a complete copy of your database. Subsequent backups are incremental and capture only the data that changed since the last backup, reducing storage costs and backup time.

InfluxDB 3 also supports **continuous backups**, which back up the database in the background at all times, enabling point-in-time restore to any moment within the retention window.

> **Note**
>
> Data written within the last 15 minutes might not be included in the backup. To recover the most recent data, use point-in-time restore with a timestamp at least 15 minutes in the past.

Key concepts

Concept	Description
DbBackup	A backup of your database containing the cluster configuration at the time of backup.
DbBackupConfiguration	Automated recurring backups at a defined frequency and retention.
Restore	Creates a new DB from a DbBackup or restores an existing resource back to the provided DbBackup.
RetentionDays	Defines the number of days a backup is kept before automatic deletion.
On Demand Backup	Triggers a one-time backup of the database at a point in time.
Continuous Backup	The database backs up in the background at all times and is available for point-in-time restore within the retention window.
Custom Schedule Backup	Defines a custom AWS cron schedule when the DB will be backed up automatically.

Creating a DbBackup

On-demand backup (AWS CLI)

Create an on-demand backup of a cluster:

```
aws timestream-influxdb create-db-backup \
    --db-resource-id my-db-cluster-id \
    --name my-db-backup-2026-07-07 \
    --retention-days 7
```

On-demand backup (AWS Management Console)

1. Open the Amazon Timestream for InfluxDB console.

2. Choose your InfluxDB 3 cluster.
3. Choose **Actions**, **Create backup**.
4. Enter a backup name and retention period.
5. Choose **Create backup**.

Configuring automated backups

Automated backups can be enabled at resource creation time or updated afterwards. Only one DbBackupConfiguration is allowed per type per resource, and a maximum of 4 DbBackupConfigurations can be enabled per resource.

Enable automated backups on create (AWS CLI)

```
aws timestream-influxdb create-db-cluster \
    --name my-db-cluster \
    --db-instance-type db.influx.medium \
    --vpc-subnet-ids subnet-0123456789abcdef0 \
    --vpc-security-group-ids sg-0123456789abcdef0 \
    --db-parameter-group-identifier InfluxDBV3Core \
    --db-backup-configurations '[{
       "enabled": true,
       "type": "CONTINUOUS",
       "retentionDays": 30
      },
      {
       "enabled": true,
       "type": "DAILY",
       "retentionDays": 7
      },
      {
       "enabled": true,
       "type": "WEEKLY",
       "retentionDays": 30
      },
      {
       "enabled": true,
       "type": "MONTHLY",
       "retentionDays": 365
      }]'
```

Enable automated backups on update (AWS CLI)

DbBackupConfigurations can be enabled, disabled, and retention can be changed after creation:

```
aws timestream-influxdb update-db-cluster \
    --db-cluster-id my-db-cluster-id \
    --db-backup-configurations '[{
       "enabled": true,
       "type": "CONTINUOUS",
       "retentionDays": 30
      },
      {
       "enabled": true,
       "type": "DAILY",
       "retentionDays": 7
      },
      {
       "enabled": false,
       "type": "WEEKLY",
       "retentionDays": 30
      }]'
```

DbBackupConfiguration types

Type	Description
CONTINUOUS	Creates a continuous backup that can be restored to any point in time within the last 15 minutes up to the creation date or retention date (whichever is later). Maximum retention is 35 days.
HOURLY	Creates a DbBackup every hour
DAILY	Creates a DbBackup every day at midnight in the timezone of the Region
WEEKLY	Creates a DbBackup every week on Sunday at midnight in the timezone of the Region
MONTHLY	Creates a DbBackup every month on the 1st at midnight in the timezone of the Region

Type	Description
CUSTOM	Use an AWS Cron Expression for a custom frequency. Must be at a daily frequency or less.

Example CUSTOM type:

```
--db-backup-configurations '[{
     "enabled": true,
     "type": "CUSTOM",
     "retentionDays": 7,
     "customSchedule": "cron(00 18 * * * *)"
    }]'
```

This creates a backup that runs at 18:00 every day in the local timezone.

> **Note**
>
> Automated backups can start anytime within an hour of the scheduled time.

Best practices for scheduling

- Use `CONTINUOUS` for maximum coverage with point-in-time restore and up to 35-day retention.
- Use longer frequencies like `WEEKLY` or `MONTHLY` with longer retention days for extended archival.

Listing and viewing backups

List all backups for a resource:

```
aws timestream-influxdb list-db-backups \
    --db-resource-id my-resource-id
```

List all backups in the account:

```
aws timestream-influxdb list-db-backups
```

Get backup details:

```
aws timestream-influxdb get-db-backup \
    --identifier my-db-backup-id
```

DbBackup statuses

Status	Description
IN_PROGRESS	Backup is being created (cluster remains available)
COMPLETED	Backup is completed and available to use for restore
FAILED	Backup creation failed
DELETING	Backup is being deleted
DELETED	Backup has been deleted

Restoring from a DbBackup

A DbBackup can be restored to a new resource, or it can replace the existing resource where the backup was taken.

Restore to a new resource

This is the default and recommended restore mode since it does not impact any existing resources.

```
aws timestream-influxdb restore-from-db-backup \
    --name my-new-restored-database \
    --db-backup-id my-backup-id \
    --restore-mode NEW_RESOURCE
```

If no additional parameters are provided, the new resource inherits all parameters from the DbBackup. A subset of these parameters can be overridden during restore if provided.

Point-in-time restore (continuous backups)

For a `CONTINUOUS` backup, you can specify a `--restore-to-time` parameter for the exact UNIX epoch timestamp in seconds to restore the database. The timestamp must be between the last 15 minutes up to the retention days or time of backup creation (whichever is later).

```
aws timestream-influxdb restore-from-db-backup \
    --name my-pitr-restored-database \
    --db-backup-id my-continuous-backup-id \
    --restore-mode NEW_RESOURCE \
    --restore-to-time 1751382000
```

Restore to an existing resource

```
aws timestream-influxdb restore-from-db-backup \
    --name my-existing-resource-name \
    --db-backup-id my-backup-id \
    --restore-mode REPLACE_EXISTING
```

When restoring to an existing resource, no additional parameters are accepted and no changes to the existing database configuration will be made besides restoring the data.

> **Warning**
>
> Restoring to an existing resource is a **destructive operation**. The resource will not be available during this operation. Existing data will be deleted and replaced with the data from the time of the backup. If there are failures during the operation, the service will attempt to roll back the resource with the original data. It is highly recommended to take an on-demand DbBackup prior to performing this action. After the operation completes, the original data is deleted and not recoverable unless a DbBackup was taken.

You can restrict this operation using an IAM policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
```

```
            "Action": "timestream-influxdb:RestoreFromDbBackup",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "timestream-influxdb:RestoreMode": "REPLACE_EXISTING"
                }
            }
        }
    ]
}
```

Restore (AWS Management Console)

1. Open the Amazon Timestream for InfluxDB console.
2. In the navigation pane, choose **Backups**.
3. Select the backup you want to restore from.
4. Choose **Actions**, **Restore backup**.
5. Choose the restore mode (**New resource** or **Replace existing**).
6. For continuous backups, optionally specify the point-in-time to restore to.
7. Configure settings as needed and choose **Restore**.

Deleting backups

```
aws timestream-influxdb delete-db-backup \
    --identifier my-db-backup-id
```

If `retentionDays` is set, the DbBackup is automatically deleted on that day.

Retaining backups when deleting a resource

When deleting a resource, by default all automated DbBackups are deleted and on-demand backups are retained. To retain automated backups on resource deletion, use the `--retain-automated-backups` flag:

```
aws timestream-influxdb delete-db-cluster \
    --db-cluster-id my-db-cluster-id \
    --retain-automated-backups
```

Retained automated backups are automatically deleted at their configured expiry day depending on the retention policy.

Performance impact

Backups on Amazon Timestream for InfluxDB v3 do not have any notable impact to the performance of the database.

The first backup may take longer since it creates a full database backup. Subsequent backups are incremental and only capture changed data.

The backup start time can be delayed due to scheduling but should generally start within one hour of the on-demand backup or scheduled time.

Relationship to service-managed backups

With customer-managed backup and restore, you have full control over your backup strategy including continuous backups for point-in-time restore:

Feature	Service-managed backups	Customer-managed backups
Control	Managed by AWS	Managed by you
Access	Via AWS support ticket	Self-service (Console, CLI, API)
Frequency	Automatic	On-demand, scheduled, or continuous
Point-in-time restore	Not available	Available with CONTINUOUS type (up to 35-day window)
Retention	Managed by AWS	Defined by you (per configuration)
Restore method	Support ticket required	Self-service restore (new resource or replace existing)

Security and encryption

- All backups are encrypted at rest using service-managed AWS KMS keys.

- For resources with customer managed keys (CMK) enabled, backups are encrypted with the same AWS KMS key.
- Access to backup operations is controlled through IAM policies.
- Backup data remains in the same AWS Region as your cluster.

Limitations

- Backups can only be restored within the same AWS account and Region where they were created.
- Cross-Region and cross-account restore is not supported.
- A maximum of 4 DbBackupConfigurations can be enabled per resource.
- Custom schedule frequency must be at a daily frequency or less.
- Continuous backup maximum retention is 35 days.
- Point-in-time restore is only available for continuous backups, not scheduled backups.
- In-place restore (`REPLACE_EXISTING` mode) is not supported for clusters encrypted with a customer managed key (CMK). Use `NEW_RESOURCE` mode instead.

Pricing

There is no additional charge for the customer-managed backup and restore feature. You pay only for the Amazon S3 storage used by your backups. Incremental backups reduce storage costs by capturing only changed data. For Amazon S3 storage pricing, see Amazon S3 pricing.

Amazon EventBridge event notifications

Amazon Timestream for InfluxDB 3 publishes events to Amazon EventBridge when your clusters undergo state changes. Events are emitted for lifecycle operations including cluster creation, deletion, compute scaling, parameter group updates, maintenance windows, node addition/removal, engine type conversion, and reboot—covering both successful completions and failures.

Events are published to the default EventBridge event bus in your account with source `aws.timestream-influxdb` and detail-type `Timestream InfluxDB DB Cluster Event`. The `SourceType` field is `DB_CLUSTER` for all InfluxDB 3 events.

You can use EventBridge rules to:

- Trigger automation when a scaling operation or node addition completes.

- Route failure events for immediate on-call alerting.
- Persist all events to Amazon CloudWatch Logs or Amazon S3 for audit trails.
- Suppress monitoring alerts during maintenance windows.

Example: Match all InfluxDB 3 cluster failure events

```
{
  "source": ["aws.timestream-influxdb"],
  "detail-type": ["Timestream InfluxDB DB Cluster Event"],
  "detail": {
    "EventCategories": ["failure"]
  }
}
```

Example: Match scaling events for a specific cluster

```
{
  "source": ["aws.timestream-influxdb"],
  "detail": {
    "SourceIdentifier": ["my-influxdb3-cluster"],
    "EventID": ["TIDB-EVENT-22001", "TIDB-EVENT-22011"]
  }
}
```

For the complete event schema, event reference tables, setup instructions, common patterns, and troubleshooting guidance, see the section called “Amazon EventBridge events”.

There is no additional charge for publishing events. Standard Amazon EventBridge pricing applies for rule evaluation and target delivery.

Writing data to your Timestream for InfluxDB 3 cluster

Amazon Timestream for InfluxDB 3 provides robust capabilities for ingesting time-series data efficiently. Understanding the proper methods for writing data is essential for maximizing performance and ensuring data integrity.

Timestream for InfluxDB 3 provides multiple HTTP API endpoints for writing time-series data, offering flexibility for different integration methods and compatibility with existing InfluxDB workloads.

Line protocol overview

InfluxDB 3 is designed for high write-throughput and uses an efficient, human-readable write syntax called line protocol. As a schema-on-write database, InfluxDB automatically creates the logical database, tables, and their schemas when you start writing data, without requiring any manual setup. Once the schema is created, InfluxDB validates future write requests against it before accepting new data, while still allowing schema evolution as your needs change.

Line protocol structure

Line protocol consists of the following essential elements:

- **Table**: A string identifier for the table where data will be stored.
- (Optional) **Tag set**: Comma-delimited key-value pairs representing metadata (indexed).
- **Field set**: Comma-delimited key-value pairs representing the actual measurements.
- (Optional) **Timestamp**: Unix timestamp associated with the data point up to nanosecond precision.

Field values can be one of the following data types:

- Strings (must be quoted)
- Floats (for example, 23.4)
- Integers (for example, 10i)
- Unsigned integers (for example, 10u)
- Booleans (true/false)

Line protocol follows this general syntax:

```
myTable,tag1=val1,tag2=val2 field1="v1",field2=1i 0000000000000000000
```

Example data point using line protocol:

```
home,room=Living\ Room temp=21.1,hum=35.9,co=0i 1735545600
```

This creates a point in the "home" table with:

- Tag: room="Living Room"

- Fields: temp=21.1 (float), hum=35.9 (float), co=0 (integer)
- Timestamp: 1735545600 (Unix seconds)

API endpoints overview

InfluxDB 3 supports three primary write endpoints:

1. **Native v3 API** (`/api/v3/write_lp`): The recommended endpoint for new implementations.
2. **v2 Compatibility API** (`/api/v2/write`): For migrating InfluxDB v2.x workloads.
3. **v1 Compatibility API** (`/write`): For migrating InfluxDB v1.x workloads.

Using the Native v3 write API

The `/api/v3/write_lp` endpoint is the native InfluxDB 3 API for writing line protocol data.

Request format:

```
POST /api/v3/write_lp?
db=DATABASE_NAME&precision=PRECISION&accept_partial=BOOLEAN&no_sync=BOOLEAN
```

Query parameters:

Parameter	Description	Default
db	Database name (required)	-
`precision`	Timestamp precision (ns, us, ms, s)	Auto-detected
`accept_partial`	Accept partial writes on errors	true
no_sync	Acknowledge before WAL persistence	false

Example write request:

```
curl -v "https://your-cluster-endpoint:8086/api/v3/write_lp?db=sensors&precision=s" \
```

```
  --header "Authorization: Bearer YOUR_TOKEN" \
  --data-raw "home,room=Living\ Room temp=21.1,hum=35.9,co=0i 1735545600
home,room=Kitchen temp=21.0,hum=35.9,co=0i 1735545600"
```

Write response modes

Standard Mode (`no_sync=false`)

- Waits for data to be written to the WAL (Write-Ahead Log) before acknowledging.
- Provides durability guarantees.
- Higher latency due to WAL persistence wait.
- Recommended for critical data where durability is essential.

Fast Mode (`no_sync=true`)

- Acknowledges immediately without waiting for WAL persistence.
- Lowest possible write latency.
- Risk of data loss if system crashes before WAL write completes.
- Ideal for high-throughput scenarios where speed is prioritized over absolute durability.

Partial write handling

The `accept_partial` parameter controls behavior when write batches contain errors:

When `accept_partial` is `true` (default):

- Valid lines are written successfully.
- Invalid lines are rejected.
- Returns 400 status with details about failed lines.
- Useful for large batch operations where some failures are acceptable.

When `accept_partial` is `false`:

- Entire batch is rejected if any line fails.
- No data is written.
- Returns 400 status with error details.

- Ensures all-or-nothing write semantics.

Compatibility APIs

Compatibility APIs enable seamless migration of existing InfluxDB v1 or v2 workloads to InfluxDB 3. These endpoints work with existing InfluxDB client libraries, Telegraf, and third-party integrations.

Important differences:

- Tags in a table (measurement) are immutable once created.
- A tag and a field cannot have the same name within a table.
- Schema validation is enforced on write.

InfluxDB v2 compatibility

The `/api/v2/write` endpoint provides backwards compatibility for v2 clients:

```
curl -i "https://your-cluster-endpoint:8086/api/v2/write?
bucket=DATABASE_NAME&precision=s" \
  --header "Authorization: Bearer DATABASE_TOKEN" \
  --header "Content-type: text/plain; charset=utf-8" \
  --data-binary 'home,room=kitchen temp=72 1641024000'
```

V2 API parameters:

Parameter	Location	Description
`bucket` *	Query string	Maps to database name
`precision`	Query string	Timestamp precision (ns, us, ms, s, m, h)
`Authorization`	Header	Bearer or Token scheme
`Content-Encoding`	Header	gzip or identity

InfluxDB v1 Compatibility

The `/write` endpoint provides backwards compatibility for v1 clients:

```
curl -i "https://your-cluster-endpoint:8086/write?db=DATABASE_NAME&precision=s" \
  --user "any:DATABASE_TOKEN" \
  --header "Content-type: text/plain; charset=utf-8" \
  --data-binary 'home,room=kitchen temp=72 1641024000'
```

V1 authentication options:

- Basic authentication: Token as password (`--user "any:TOKEN"`).
- Query parameter: `p=TOKEN` in URL.
- Bearer/Token header: Standard authorization header.

V1 API parameters:

Parameter	Location	Description
db *	Query string	Database name
`precision`	Query string	Timestamp precision
p	Query string	Token for query auth
u	Query string	Username (ignored)
`Authorization`	Header	Multiple schemes supported
`Content-Encoding`	Header	gzip or identity

Client libraries and integrations

Official InfluxDB 3 client libraries

InfluxDB 3 client libraries provide native language interfaces for constructing and writing time-series data:

- **Python**: `influxdb3-python`
- **Go**: `influxdb3-go`
- **JavaScript/Node.js**: `influxdb3-js`

- **Java**: `influxdb3-java`
- **C#**: `InfluxDB3.Client`

Example: Python client

```
from influxdb3 import InfluxDBClient3

client = InfluxDBClient3(
    host="your-cluster-endpoint:8086",
    token="YOUR_TOKEN",
    database="DATABASE_NAME"
)

# Write using line protocol
client.write("home,room=Living\\ Room temp=21.1,hum=35.9,co=0i")

# Write using Point objects
from influxdb3 import Point
point = Point("home") \
    .tag("room", "Living Room") \
    .field("temp", 21.1) \
    .field("hum", 35.9) \
    .field("co", 0)

client.write(point)
```

Example: Go client

```
import "github.com/InfluxCommunity/influxdb3-go/v2/influxdb3"

client, err := influxdb3.New(influxdb3.ClientConfig{
    Host: "your-cluster-endpoint:8086",
    Token: "YOUR_TOKEN",
    Database: "DATABASE_NAME",
})

point := influxdb3.NewPoint("home",
    map[string]string{"room": "Living Room"},
    map[string]any{
        "temp": 24.5,
        "hum":  40.5,
        "co":   15,
```

```
    },
    time.Now(),
)

err = client.WritePoints(context.Background(), []*influxdb3.Point{point})
```

Legacy client libraries

For existing v1 and v2 workloads, you can continue using legacy client libraries with the compatibility endpoints:

Example: Node.js v1 client:

```
const Influx = require('influx')

const client = new Influx.InfluxDB({
  host: 'your-cluster-endpoint',
  port: 8086,
  protocol: 'https',
  database: 'DATABASE_NAME',
  username: 'ignored',
  password: 'DATABASE_TOKEN'
})
```

Telegraf integration with Timestream for InfluxDB 3

Telegraf is a plugin-based data collection agent with over 300 input plugins for collecting metrics from various sources and output plugins for writing data to different destinations. Its "plug-and-play" architecture makes it ideal for quickly collecting and reporting metrics to InfluxDB 3.

Requirements

- Telegraf 1.9.2 or greater – For installation instructions, see the Telegraf Installation documentation.
- InfluxDB 3 cluster endpoint and credentials.
- Network connectivity to your InfluxDB 3 cluster.

Telegraf configuration options

Telegraf provides two output plugins compatible with InfluxDB 3:

1. `outputs.influxdb_v2` - Recommended for new deployments.
2. `outputs.influxdb` (v1) - For existing v1 configurations.

Using the v2 output plugin

We recommend that you use the `outputs.influxdb_v2` plugin to connect to the InfluxDB v2 compatibility API:

```
[[outputs.influxdb_v2]]
  urls = ["https://your-cluster-endpoint:8086"]
  token = "${INFLUX_TOKEN}"  # Use environment variable for security
  organization = ""           # Can be left empty for InfluxDB 3
  bucket = "DATABASE_NAME"

  ## Optional: Enable gzip compression
  content_encoding = "gzip"

  ## Optional: Increase timeout for high-latency networks
  timeout = "10s"

  ## Optional: Configure batching
  metric_batch_size = 5000
  metric_buffer_limit = 50000
```

Using the legacy v1 output plugin

For existing Telegraf configurations using the v1 plugin:

```
[[outputs.influxdb]]
  urls = ["https://your-cluster-endpoint:8086"]
  database = "DATABASE_NAME"
  skip_database_creation = true
  username = "ignored"           # Required but ignored
  password = "${INFLUX_TOKEN}"   # Use environment variable
  content_encoding = "gzip"

  ## Optional: Configure write parameters
  timeout = "10s"
  metric_batch_size = 5000
  metric_buffer_limit = 50000
```

Basic Telegraf configuration Example

The following is a complete example that collects system metrics and writes them to InfluxDB 3:

```
# Global Agent Configuration
[agent]
  interval = "10s"
  round_interval = true
  metric_batch_size = 5000
  metric_buffer_limit = 50000
  collection_jitter = "0s"
  flush_interval = "10s"
  flush_jitter = "0s"
  precision = "s"
  hostname = ""
  omit_hostname = false

# Input Plugins - Collect system metrics
[[inputs.cpu]]
  percpu = true
  totalcpu = true
  collect_cpu_time = false
  report_active = false

[[inputs.disk]]
  ignore_fs = ["tmpfs", "devtmpfs", "devfs", "iso9660", "overlay", "aufs", "squashfs"]

[[inputs.mem]]

[[inputs.net]]
  interfaces = ["eth*", "en*"]

[[inputs.system]]

# Output Plugin - Write to InfluxDB 3
[[outputs.influxdb_v2]]
  urls = ["https://your-cluster-endpoint:8086"]
  token = "${INFLUX_TOKEN}"
  organization = ""
  bucket = "telegraf_metrics"
  content_encoding = "gzip"
```

Best practices for Telegraf with InfluxDB 3

- **Security**
  - Store tokens in environment variables or secret stores.
  - Never hardcode tokens in configuration files.
  - Use HTTPS endpoints for production deployments.
- **Performance optimization**
  - Enable gzip compression with content_encoding = "gzip".
  - Configure appropriate batch sizes (5000-10000 metrics).
  - Set buffer limits based on available memory.
  - Use precision appropriate for your use case (seconds often sufficient).
- **Network configuration**
  - For private clusters, run Telegraf within the same VPC.
  - Configure appropriate timeouts for your network latency.
  - Use the writer/reader endpoint for write operations.
- **Monitoring**
  - Enable Telegraf's internal metrics plugin to monitor agent performance.
  - Monitor write errors and retries.
  - Set up alerts for buffer overflow conditions.
- **Data organization**
  - Use consistent tag naming across input plugins.
  - Leverage Telegraf's processor plugins to normalize data.
  - Apply tag filtering to control cardinality.

Running Telegraf

To start Telegraf with your configuration, do the following:

```
# Test configuration
telegraf --config telegraf.conf --test

# Run Telegraf
telegraf --config telegraf.conf
```

```
# Run as a service (systemd)
sudo systemctl start telegraf
```

Common Telegraf plugins for time series data

Popular input plugins:

- `inputs.cpu`, `inputs.mem`, `inputs.disk` - System metrics.
- `inputs.docker`, `inputs.kubernetes` - Container metrics.
- `inputs.prometheus` - Scrape Prometheus endpoints.
- `inputs.snmp` - Network device monitoring.
- `inputs.mqtt_consumer` - IoT data collection.
- `inputs.http_listener_v2` - HTTP webhook receiver.

Useful processor plugins:

- `processors.regex` - Transform tag/field names.
- `processors.converter` - Change field data types.
- `processors.aggregator` - Aggregate metrics.
- `processors.filter` - Filter metrics based on conditions.

By leveraging Telegraf's extensive plugin ecosystem with InfluxDB 3, you can build comprehensive monitoring solutions that collect data from diverse sources and efficiently write it to your time-series database.

Best practices for writing data

When writing data, we recommend the following:

- Batch optimization
  - Optimal batch size: 5,000-10,000 lines or 10MB per request.
  - Use compression (gzip) for large payloads.
  - Sort tags by key in lexicographic order for better performance.
- Timestamp precision
  - Use the coarsest precision that meets your needs.

  - Explicitly specify precision to avoid ambiguity.
  - Maintain consistent precision across your application.
- Error handling
  - Implement retry logic for transient failures.
  - Use accept_partial=true for resilient batch operations.
  - Monitor write errors through CloudWatch metrics.
- Performance tuning
  - Use no_sync=true for high-throughput scenarios.
  - Distribute writes across multiple connections.
  - Use the writer/reader endpoint for all write operations.
- Schema considerations
  - Tags are immutable once created.
  - Fields and tags cannot share the same nam.e
  - Design schemas with query patterns in mind.
  - Keep tag cardinality under control.

Important differences from previous versions:

- Immutable tags: Once a tag is created in a table, its type cannot be changed
- No tag/field name conflicts: A tag and field cannot have the same name within a table
- Schema-on-write: InfluxDB 3 validates data types on write
- Automatic table creation: Tables are created automatically on first write
- Strict type checking: Field types must remain consistent across all writes

By leveraging the appropriate write API and following these best practices, you can efficiently ingest time-series data into your Timestream for InfluxDB 3 instance while maintaining high performance and data integrity.

Querying data from Timestream for InfluxDB 3

Amazon Timestream for InfluxDB 3 provides multiple query APIs and protocols to retrieve your time series data. The platform supports both SQL and InfluxQL query languages through HTTP and Flight (gRPC) protocols, offering flexibility for different use cases and client preferences.

Query methods overview

InfluxDB 3 supports the following query methods:

- Native v3 HTTP API – SQL and InfluxQL queries via REST endpoints.
- influxdb3 CLI – Command-line interface for interactive queries.
- Flight+gRPC Protocol – High-performance binary protocol for client libraries.
- v1 Compatibility API – Legacy InfluxQL queries for backward compatibility.

Using the v3 HTTP query API

The v3 API provides dedicated endpoints for SQL and InfluxQL queries with support for both GET and POST methods.

SQL queries (`/api/v3/query_sql`)

GET request example:

```
curl --get "https://your-cluster-endpoint:8086/api/v3/query_sql" \
  --header "Authorization: Bearer YOUR_TOKEN" \
  --data-urlencode "db=DATABASE_NAME" \
  --data-urlencode "q=SELECT * FROM home WHERE time >= now() - INTERVAL '1 day'"
```

POST request example (for complex queries):

```
curl "https://your-cluster-endpoint:8086/api/v3/query_sql" \
  --header "Authorization: Bearer YOUR_TOKEN" \
  --json '{
    "db": "DATABASE_NAME",
    "q": "SELECT * FROM home WHERE room = '\''Kitchen'\'' AND temp > 20",
    "format": "jsonl"
  }'
```

InfluxQL queries (`/api/v3/query_influxql`)

For InfluxQL queries do the following:

```
curl --get "https://your-cluster-endpoint:8086/api/v3/query_influxql" \
  --header "Authorization: Bearer YOUR_TOKEN" \
  --data-urlencode "db=DATABASE_NAME" \
```

```
  --data-urlencode "q=SELECT mean(temp) FROM home WHERE time >= now() - 1d GROUP BY
 room"
```

Query parameters

You can use the following query parameters:

Parameter	Description	Required
db	Database name	Yes
q	Query string (SQL or InfluxQL)	Yes
`format`	Response format (json, jsonl, csv, pretty, parquet)	No (default: json)
`params`	Parameters for parameterized queries	No

Using the Influxdb3 CLI

InfluxDB 3 command-line interface (CLI) is invoked with the `influxdb3` command. It provides an interactive way to query your data with support for multiple output formats.

The following shows a basic query:

```
influxdb3 query \
  --host "your-cluster-endpoint:8086" \
  --token "YOUR_TOKEN" \
  --database "DATABASE_NAME" \
  "SELECT * FROM home WHERE time >= now() - INTERVAL '1 day'"
```

The following shows a query with different output formats:

```
# JSON output
influxdb3 query \
  --database "DATABASE_NAME" \
  --format json \
  "SELECT * FROM home LIMIT 10"

# CSV output
influxdb3 query \
```

```
  --database "DATABASE_NAME" \
  --format csv \
  "SELECT * FROM home LIMIT 10"

# Parquet file output
influxdb3 query \
  --database "DATABASE_NAME" \
  --format parquet \
  --output results.parquet \
  "SELECT * FROM home"
```

The following output formats are supported:

- pretty (default) – Human-readable tabular format.
- json – Standard JSON array.
- jsonl – JSON Lines (streaming-friendly).
- csv – Comma-separated values.
- parquet – Binary columnar format (requires file output).

Using the v1 compatibility API

For legacy InfluxQL queries, use the `/query` endpoint:

```
curl --get "https://your-cluster-endpoint:8086/query" \
  --header "Authorization: Bearer YOUR_TOKEN" \
  --data-urlencode "db=DATABASE_NAME" \
  --data-urlencode "q=SELECT * FROM home" \
  --data-urlencode "epoch=ms"
```

Authentication options for the v3 HTTP Query API

- Bearer token: `Authorization: Bearer TOKEN`
- Basic auth: `--user "any:TOKEN"`
- Query parameter: `?p=TOKEN`

Response formats for the v3 HTTP Query API

- JSON (default)

- CSV (with `Accept: application/csv` header)

SQL Query Examples

The following shows a basic query:

```
-- Select specific fields with time filter
SELECT temp, humidity, room
FROM home
WHERE time >= now() - INTERVAL '7 days'
  AND room = 'Kitchen'
ORDER BY time DESC

-- Show all tables in database
SHOW TABLES

-- Show columns in a table
SHOW COLUMNS FROM home
```

The following shows aggregation queries:

```
-- Aggregate by tags
SELECT
  room,
  AVG(temp) as avg_temp,
  MAX(humidity) as max_humidity
FROM home
WHERE time >= now() - INTERVAL '24 hours'
GROUP BY room

-- Time-based aggregation using DATE_BIN
SELECT
  DATE_BIN(INTERVAL '1 hour', time) as time,
  AVG(temp) as avg_temp,
  COUNT(*) as reading_count
FROM home
GROUP BY 1
ORDER BY time DESC
```

The following shows advanced SQL features:

```
-- Parameterized queries (via API)
```

```
SELECT * FROM home
WHERE room = $room
  AND temp > $min_temp
  AND time >= $start_time

-- Gap filling with interpolation
SELECT
  date_bin_gapfill(INTERVAL '5 minutes', time) as time,
  room,
  interpolate(avg(temp)) as temp
FROM home
WHERE time >= '2025-01-01T00:00:00Z'
  AND time <= '2025-01-01T12:00:00Z'
GROUP BY 1, room

-- Type casting
SELECT
  temp::INTEGER as temp_int,
  CAST(humidity AS VARCHAR) as humidity_str
FROM home
```

The following shows InfluxQL query examples:

```
-- Basic query with time filter
SELECT * FROM home WHERE time >= now() - 1d

-- Aggregation with GROUP BY time
SELECT MEAN(temp), MAX(humidity)
FROM home
WHERE time >= now() - 7d
GROUP BY time(1h), room

-- Using selector functions
SELECT FIRST(temp), LAST(temp)
FROM home
WHERE time >= now() - 1h
GROUP BY room
```

The following shows how to query system tables to understand your database structure and monitor performance:

```
-- View all tables with schema information
SELECT * FROM information_schema.tables
```

```
-- View column details for a specific table
SELECT * FROM information_schema.columns
WHERE table_name = 'home'

-- Monitor recent queries
SELECT * FROM system.queries
ORDER BY issue_time DESC
LIMIT 10

-- Check cache configurations
SELECT * FROM system.last_caches
SELECT * FROM system.distinct_caches
```

Query performance best practices

- **Use Time Filters**: Always include time range filters to limit data scanned.
- **Leverage Indexes**: Design queries to use tag filters effectively.
- **Choose Appropriate Output Format**:
  - Use jsonl for streaming large results.
  - Use parquet for data export and analytics.
  - Use csv for spreadsheet compatibility.
- **Optimize Aggregations**: Use DATE_BIN for time-based grouping.
- **Use Parameterized Queries**: Prevent injection attacks and improve reusability.
- **Monitor Query Performance**: Check the system.queries table for slow queries.

Client library query support

InfluxDB 3 client libraries use the Flight+gRPC protocol for optimal performance. The following Python code shows an example of this.

```
from influxdb3 import InfluxDBClient3

client = InfluxDBClient3(
    host="your-cluster-endpoint:8086",
    token="YOUR_TOKEN",
    database="DATABASE_NAME"
)
```

```
# SQL query
sql_result = client.query("SELECT * FROM home WHERE room = 'Kitchen'")

# InfluxQL query
influxql_result = client.query(
    "SELECT MEAN(temp) FROM home WHERE time >= now() - 1h GROUP BY room",
    language="influxql"
)
```

Compare SQL and InfluxQL query capabilities

The following table compares SQL and InfluxQL query capabilities:

Feature	SQL	InfluxQL
Joins	Supported	Not supported
Window Functions	Full support	Limited
Subqueries	Supported	Not supported
Time Functions	`DATE_BIN`, `INTERVAL`	`time()` grouping
Parameterized Queries	Native support	Not supported
Gap Filling	`date_bin_gapfill()`	`fill()` function

By understanding these query capabilities and choosing the appropriate method for your use case, you can efficiently retrieve and analyze your time-series data from Timestream for InfluxDB 3.

SQL advantages:

- Full-featured SQL implementation with support for complex queries.
- Support for joins, unions, and window functions.
- Familiar syntax for users with SQL background.
- More extensive analytical capabilities.

InfluxQL advantages:

- Designed specifically for time-series data.
- Simpler syntax for common time-series operations.
- Backward compatibility for users migrating from InfluxDB v1.
- Specialized time-series functions.

Query optimization features

InfluxDB 3 offers several optimization features to improve query performance for specific use cases. These features leverage in-memory caching and custom indexing strategies to deliver sub-millisecond response times for frequently accessed data patterns.

Last Value Cache (LVC)

The Last Value Cache (LVC) stores the most recent N values for specified fields in memory, enabling sub-10ms response times for queries that need the latest data points. This feature is available in both Core and Enterprise editions.

How LVC works

The LVC maintains an in-memory table of the most recent values for each unique combination of key columns (typically tags). For example, with sensor data:

```
Table: home
### Tags: room, wall
### Fields: temp, humidity, co

LVC Configuration:
- Key columns: room, wall
- Value columns: temp, humidity, co
- Count: 4 (last 4 values)
```

The cache stores:

room	wall	temp	humidity	co	time
Kitchen	east	22.7	36.5	26	2025-01-26T20:00:00Z
Kitchen	east	22.7	36.0	9	2025-01-26T17:00:00Z

room	wall	temp	humidity	co	time
Kitchen	east	22.7	36.2	3	2025-01-26T15:00:00Z
Kitchen	east	22.7	36.1	0	2025-01-26T10:00:00Z
Living Room	north	22.2	36.4	17	2025-01-26T20:00:00Z
...	...	...	...	...	...

Create an LVC

Use the following command to create an LVC:

```
influxdb3 create last_cache \
  --database DATABASE_NAME \
  --token YOUR_TOKEN \
  --table home \
  --key-columns room,wall \
  --value-columns temp,hum,co \
  --count 5 \
  --ttl 30mins \
  homeLastCache
```

Query the LVC

Use the following command to query an LVC:

```
-- Query the last cached values
SELECT * FROM last_cache('home', 'homeLastCache')

-- Filter specific series
SELECT * FROM last_cache('home', 'homeLastCache')
WHERE room = 'Kitchen'
```

LVC best practices

- Manage Cardinality: Only include essential tags as key columns to minimize memory usage.
- Optimize Value Count: Balance between query needs and memory consumption.
- Consider TTL: Set appropriate time-to-live for cache entries.

- Monitor Memory: Cache size = (key_column_cardinality × count × value_columns).

Distinct Value Cache (DVC)

The Distinct Value Cache (DVC) maintains unique values for specified columns in memory, accelerating metadata queries to under 30ms response times. Available in both Core and Enterprise editions.

How DVC works

The DVC stores all unique combinations of values for specified columns, perfect for queries that need to list available tag values or metadata.

Example cache for location data:

country	county	city
Austria	Salzburg	Salzburg
Austria	Vienna	Vienna
Belgium	Antwerp	Antwerp
Belgium	West Flanders	Bruges
Czech Republic	Prague	Prague

Create a DVC

Use the following command to create a DVC:

```
influxdb3 create distinct_cache \
  --database DATABASE_NAME \
  --token YOUR_TOKEN \
  --table wind_data \
  --columns country,county,city \
  --max-cardinality 10000 \
  --max-age 24h \
  windDistinctCache
```

Query the DVC

Use the following command to query a DVC:

```
-- Get all distinct values
SELECT * FROM distinct_cache('wind_data', 'windDistinctCache')

-- Get distinct countries
SELECT DISTINCT country
FROM distinct_cache('wind_data', 'windDistinctCache')
```

DVC best practices

- Set Cardinality Limits: Define maximum unique value combinations to control memory.
- Configure Max Age: Remove stale values automatically.
- Cache Strategic Columns: Focus on columns frequently used in metadata queries.
- Monitor Cache Size: Higher cardinality means that more memory is required.

File indexes available in Enterprise only

Available in Enterprise Edition only. File indexes allow customization of how data is indexed in storage, significantly improving single-series query performance.

Default vs. custom indexing

Default Indexing: InfluxDB indexes on all tags plus time.

Custom Indexing: Index only on columns relevant to your queries.

Example optimization:

```
Schema columns: country, state_province, county, city, postal_code
Query patterns: Always filter by country, state_province, city
Custom index: time, country, state_province, city (skip county, postal_code)
```

The following shows how to create a custom file index

```
influxdb3 create file_index \
  --database DATABASE_NAME \
```

```
  --token YOUR_TOKEN \
  --table wind_data \
  country,state_province,city
```

The following shows how to delete a file index:

```
influxdb3 delete file_index \
  --database DATABASE_NAME \
  --token YOUR_TOKEN \
  --table wind_data
```

File index considerations

- Compaction Required: Indexes are built during data compaction (gen2+).
- String Columns Only: Can index both tags and string fields.
- Query Pattern Analysis: Analyze your workload before creating custom indexes.
- Single-Series Optimization: Most beneficial for queries targeting specific series.

Managing cache information

View cache configurations and statistics using system tables:

```
-- View Last Value Caches
SELECT * FROM system.last_caches

-- View Distinct Value Caches
SELECT * FROM system.distinct_caches

-- Check cache memory usage
SELECT
  cache_name,
  table_name,
  key_columns,
  value_count,
  memory_size_bytes
FROM system.last_caches
```

Performance optimization strategy

Choose the right optimization features based on your query patterns:

Query Pattern	Recommended Feature	Expected Performance
Latest values per series	Last Value Cache	<10ms
Available tag values	Distinct Value Cache	<30ms
Single-series queries	File Indexes (Enterprise)	Significant improvement
Time-range aggregations	Standard indexes	Baseline performance

Memory and resource considerations

Cache memory formula:

- LVC: memory = key_cardinality × value_count × value_columns × data_size
- DVC: memory = distinct_combinations × column_count × data_size

Best practices:

- Start with small caches and monitor memory usage
- Use TTL settings to limit cache growth
- Only cache frequently queried data
- Monitor cache hit rates through system tables
- For Enterprise: Combine caches with custom file indexes for optimal performance

By leveraging these optimization features appropriately, you can achieve significant performance improvements for specific query patterns while managing resource consumption effectively.

Schema design recommendations for Timestream for InfluxDB 3

Learn how to design schemas in InfluxDB 3 Enterprise to optimize query performance and simplify data management in Amazon Timestream.

Topics

- InfluxDB data structure in Timestream for InfluxDB 3
- Schema design best practices for Timestream for InfluxDB 3

InfluxDB data structure in Timestream for InfluxDB 3

InfluxDB 3 Enterprise organizes time-series data into databases and tables:

- Database: A named location where time-series data is stored (equivalent to a bucket in InfluxDB Cloud Serverless).
- Table: A logical grouping for time-series data. (equivalent to a measurement in InfluxDB Cloud Serverless)
- Tags: Key-value pairs storing metadata as string values.
- Fields: Key-value pairs storing actual measurement data.
- Timestamp: Nanosecond-scale UTC Unix timestamp associated with each data point .

Primary keys

The primary key for a row is the combination of the point's timestamp and tag set. The primary key does not include tags with null values.

Tags and fields

- Tags: Store metadata or identifying information about the data source.
  - Can only be string values.
  - Used for filtering and grouping data.
- Fields: Store measured values.
  - Support multiple data types: Integer, Unsigned integer, Float, String, Boolean.

Schema restrictions

- No Duplicate Names: Tags and fields within the same table must have unique names.
- Column Limit: Tables have a maximum number of columns (including time column).

Schema design best practices for Timestream for InfluxDB 3

By following these guidelines, you can design InfluxDB schemas that enable simpler and more performant queries while optimizing resource utilization.

Performance optimization guidelines

- Sort tags by query priority:
  - The first write to a table determines physical column order in storage.
  - Place most frequently queried tags first for better performance.
  - Column order cannot be changed after initial write.
- Avoid wide schemas:
  - Limit the number of columns (tags and fields) per table.
  - Too many columns can increase resource usage and reduce performance.
  - Consider segmenting fields into separate tables if needed.
- Avoid sparse schemas:
  - Sparse schemas contain many null values across rows.
  - Caused by non-homogenous table schemas or writing individual fields with different timestamps.
  - Adds unnecessary overhead to storing and querying data.
- Maintain homogenous table schemas:
  - Each row should have the same tag and field keys.
  - Avoid tables with many null values.
- Use appropriate data types:
  - Use the most appropriate data type for your data.
  - Integer and boolean fields outperform string fields in queries.
- Use last-value and distinct-value cache for repetitive queries where possible.

Query simplicity guidelines

- Keep N=names simple:
  - Use one tag or field for each data attribute.
  - Choose descriptive, simple names for tables, tags, and fields.

  - Avoid embedding multiple data attributes in a single name.
- Avoid keywords and special characters:
  - Do not use SQL or InfluxQL reserved keywords.
  - Avoid special characters in table names, tag keys, and field keys.
  - Using keywords or special characters requires additional quoting in queries.

Extend Timestream for InfluxDB with processing engine plugins

The Processing engine is an embedded Python virtual machine that runs inside your InfluxDB 3 database in Amazon Timestream. It's available in both Core and Enterprise editions. It enables you to extend your database with custom Python code that can automate workflows, transform data, and create custom API endpoints.

The Processing engine executes Python plugins in response to specific database events:

- **Data writes**: Process and transform data as it enters the database
- **Scheduled events**: Run code at defined intervals or specific times
- **HTTP requests**: Expose custom API endpoints that execute your code

The engine includes an in-memory cache for managing state between executions, enabling you to build stateful applications directly within your database.

InfluxData certified plugins

At launch, InfluxDB 3 includes a set of pre-built, fully configurable plugins certified by InfluxData:

- **Data transformation**: Process and enrich incoming data
- **Alerting**: Send notifications based on data thresholds
- **Aggregation**: Calculate statistics on time-series data
- **System monitoring**: Track resource usage and health metrics
- **Integration**: Connect to external services and APIs

These certified plugins are ready to use and can be configured through trigger arguments to meet your specific requirements.

> **Important**
>
> Custom plugins are **not** currently supported. Only InfluxData certified plugins listed in this documentation can be deployed. You cannot upload or execute your own Python code as plugins at this time.

Plugin types and trigger specifications

Plugin Type	Trigger Specification	When Plugin Runs	Use Cases
Data write	`table:<TABLE_NAME>` or `all_tables`	When data is written to tables	Data transformation, alerting, derived metrics
Scheduled	`every:<DURATION>` or `cron:<EXPRESSION>`	At specified intervals	Periodic aggregation, reports, health checks
HTTP request	`request:<REQUEST_PATH>`	When HTTP requests are received	Custom APIs, webhooks, user interfaces

Create triggers

Triggers connect plugins to database events and define when they execute. Use the `influxdb3 create trigger` command.

To create a data write trigger:

```
# Trigger on writes to a specific table
influxdb3 create trigger \
  --trigger-spec "table:sensor_data" \
  --plugin-filename "process_sensors.py" \
  --database DATABASE_NAME \
  sensor_processor

# Trigger on all table writes
```

```
influxdb3 create trigger \
  --trigger-spec "all_tables" \
  --plugin-filename "process_all_data.py" \
  --database DATABASE_NAME \
  all_data_processor
```

To create a scheduled trigger:

```
# Run every 5 minutes
influxdb3 create trigger \
  --trigger-spec "every:5m" \
  --plugin-filename "periodic_check.py" \
  --database DATABASE_NAME \
  regular_check

# Run daily at 8am (cron format with seconds)
influxdb3 create trigger \
  --trigger-spec "cron:0 0 8 * * *" \
  --plugin-filename "daily_report.py" \
  --database DATABASE_NAME \
  daily_report
```

To create an HTTP request trigger:

```
# Create endpoint at /api/v3/engine/webhook
influxdb3 create trigger \
  --trigger-spec "request:webhook" \
  --plugin-filename "webhook_handler.py" \
  --database DATABASE_NAME \
  webhook_processor
```

Access the endpoint at: `https://your-cluster-endpoint:8086/api/v3/engine/webhook`

Configure triggers

Passing arguments to plugins

Configure plugin behavior using trigger arguments:

```
influxdb3 create trigger \
  --trigger-spec "every:1h" \
```

```
  --plugin-filename "threshold_check.py" \
  --trigger-arguments "threshold=90,notify_email=admin@example.com" \
  --database DATABASE_NAME \
  threshold_monitor
```

Arguments are passed to the plugin as a dictionary:

```
def process_scheduled_call(influxdb3_local, call_time, args=None):
    if args and "threshold" in args:
        threshold = float(args["threshold"])
        email = args.get("notify_email", "default@example.com")
        # Use arguments in your logic
```

Error handling behavior

Configure how triggers handle errors:

```
# Log errors (default)
influxdb3 create trigger \
  --trigger-spec "table:metrics" \
  --plugin-filename "process.py" \
  --error-behavior log \
  --database DATABASE_NAME \
  log_processor

# Retry on error
influxdb3 create trigger \
  --trigger-spec "table:critical_data" \
  --plugin-filename "critical.py" \
  --error-behavior retry \
  --database DATABASE_NAME \
  retry_processor

# Disable trigger on error
influxdb3 create trigger \
  --trigger-spec "request:webhook" \
  --plugin-filename "webhook.py" \
  --error-behavior disable \
  --database DATABASE_NAME \
  auto_disable_processor
```

Asynchronous execution

Allow multiple trigger instances to run simultaneously:

```
influxdb3 create trigger \
  --trigger-spec "table:metrics" \
  --plugin-filename "heavy_process.py" \
  --run-asynchronous \
  --database DATABASE_NAME \
  async_processor
```

Manage triggers

To view triggers for a database:

```
# Show all triggers for a database
influxdb3 show summary \
  --database DATABASE_NAME \
  --token YOUR_TOKEN
```

Table exclusion for write triggers

To filter tables within your plugin code when using `all_tables`:

```
influxdb3 create trigger \
  --trigger-spec "all_tables" \
  --plugin-filename "processor.py" \
  --trigger-arguments "exclude_tables=temp_data,debug_info" \
  --database DATABASE_NAME \
  data_processor
```

The plugin implementation is as follows:

```
def process_writes(influxdb3_local, table_batches, args=None):
    excluded_tables = set(args.get('exclude_tables', '').split(','))

    for table_batch in table_batches:
        if table_batch["table_name"] in excluded_tables:
            continue
        # Process allowed tables
```

Distributed deployment considerations

In multi-node deployments, configure plugins based on node roles:

Plugin Type	Node Type	Reason
Data write plugins	Ingester nodes	Process data at ingestion point
HTTP request plugins	Querier nodes	Handle API traffic
Scheduled plugins	Any configured node	Can run on any node with scheduler

The following considerations are important for enterprise deployments:

- Maintain identical plugin configurations across all relevant nodes.
- Route external clients (Grafana, dashboards) to querier nodes.
- Ensure plugins are available on nodes where their triggers execute.

Best practices

- **Plugin configuration**
  - Use trigger arguments for configurable values instead of hardcoding.
  - Implement proper error handling within plugins.
  - Use the `influxdb3_local` API for database operations.
- **Performance optimization**
  - Use asynchronous execution for heavy processing tasks.
  - Implement early returns for filtered data.
  - Minimize database queries within plugins.
- **Error management**
  - Choose appropriate error behavior (log, retry, or disable).
  - Monitor plugin execution through system tables.
  - Test plugins thoroughly before production deployment.
- **Security considerations**
  - Validate all input data in HTTP request plugins.

- Use secure methods for storing sensitive configuration.
- Limit plugin permissions to required operations only.

Monitor plugin execution

Query system tables to monitor plugin performance:

```
-- View processing engine logs
SELECT * FROM system.processing_engine_logs
WHERE time > now() - INTERVAL '1 hour'
ORDER BY time DESC

-- Check trigger status
SELECT * FROM system.processing_engine_triggers
WHERE database = 'DATABASE_NAME'
```

The Processing engine provides a powerful way to extend InfluxDB 3 functionality while keeping your data processing logic close to your data, reducing latency and simplifying your architecture.

InfluxData certified plugins

Amazon Timestream for InfluxDB 3 includes a comprehensive set of pre-built, certified plugins that extend database functionality without requiring custom development. These plugins are fully configurable and ready to use at launch, providing advanced capabilities for data processing, monitoring, and alerting.

For complete documentation and source code, visit the [InfluxData Plugins Repository](#).

Available plugins

Timestream for LiveAnalytics migration plugin

Migrate a database from Timestream for LiveAnalytics to Timestream for InfluxDB

- **Trigger types**: HTTP
- **Use cases**: Migrating a time series database from Timestream for LiveAnalytics to Timestream for InfluxDB.
- **GitHub**: [Timestream for LiveAnalytics migration plugin Documentation](#)

How it works: The Timestream for LiveAnalytics migration plugin works in unison with the Timestream for LiveAnalytics migration client. The client executes a Timestream for LiveAnalytics UNLOAD command to export a LiveAnalytics database to an S3 bucket in Parquet format. After the data is exported the client generates presigned URLs for the Parquet files, and invokes the migration plugin with the presigned URLs. During plugin execution the S3 objects are retrieved from the S3 bucket and transformed to InfluxDB line protocol, and written to an InfluxDB 3 database.

Best practices: The Timestream for LiveAnalytics migration plugin must be run on a single InfluxDB 3 Enterprise node. Ensure the InfluxDB 3 endpoint used with the plugin client is a process node endpoint rather than the cluster endpoint. The cluster performing the migration should not undertake ingestion or queries while the migration plugin is running, as this could lead to out of memory errors.

Migration performance depends on the resources available to the InfluxDB 3 node and the characteristics of the data being migrated. In our testing, we observed a throughput of 30,000,000 LiveAnalytics records migrated per hour. Your actual performance may vary based on several factors.

Data mapping: The following table shows how Timestream for LiveAnalytics data is mapped to line protocol data.

Timestream for LiveAnalytics Concept	Line Protocol Concept
Table	Measurement
Dimensions	Tags
Measure name	Tag
Measures	Fields
Time	Timestamp

Single-measure record transformation: The following is a single-measure record in Timestream for LiveAnalytics in the table `example_table`:

host	region	request_id	measure_name	time	measure_value::double
host1	us-west-2	saio3242ovnfk	cpu_usage	2025-04-17 16:42:54.702394001	0.66

This record will be transformed to:

```
example_table,host=host1,region=us-
west-2,request_id=saio3242ovnfk,measure_name=cpu_usage measure_value::double=0.66
 1744908174702394001
```

Multi-measure record transformation: The following is a multi-measure record in Timestream for LiveAnalytics in the table `example_table` with everything to the right of `time` being measures:

host	region	request_id	measure_name	time	cpu_usage	memory_usage
host1	us-west-2	saio3242ovnfk	metrics	2025-04-17 16:42:54.702394001	0.66	0.21

This record will be transformed to:

```
example_table,host=host1,region=us-west-2,request_id=saio3242ovnfk,measure_name=metrics
 cpu_usage=0.66,memory_usage=0.21 1744908174702394001
```

Important

The presigned URLs that the plugin uses to retrieve the LiveAnalytics data in S3 expire when either their set expiration occurs or the IAM credentials used to generate them expire (maximum 7 days). We recommend running the migration client on an EC2 instance (a `t3.medium` instance is sufficient) because the EC2 instance rotates IAM credentials automatically, removing the presigned URL time constraints during the migration. If you

do not use an EC2 instance, migrations can be resumed and large datasets may require multiple resume invocations.
The Timestream for LiveAnalytics migration plugin is recommended for migrations with under 1 billion records or 125GB within a single LiveAnalytics database.
The migration plugin should only be used on a single process node in the cluster. You can determine the process node by using list-db-instances-for-cluster and setting the `INFLUXDB3_HOST_URL` to the endpoint of one of the database instances that has an instance mode of type `PROCESS`, or you can use the Timestream console and select your cluster to find the process node.

Key features:

- Exports the time series data from Timestream for LiveAnalytics to an S3 bucket using the UNLOAD command.
- Generates presigned URLs for each S3 object being migrated.
- Tracks the migration process for each S3 object.
- Cleans up S3 objects after a successful migration.
- Supports resuming a failed migration in the event of presigned URL expiration.

Example usage:

```
# Migrate a LiveAnalytics database to InfluxDB 3
export INFLUXDB3_HOST_URL="https://<your InfluxDB 3 URL>:<your InfluxDB 3 port>"
export INFLUXDB3_AUTH_TOKEN="<your InfluxDB 3 token>"
export INFLUXDB3_DATABASE_NAME="<your InfluxDB 3 target database>"

aws s3api create-bucket --bucket <your S3 bucket name> \
    --object-lock-enabled-for-bucket --region <your region> \
    --create-bucket-configuration LocationConstraint=<your region>
```

Note

Update the S3 bucket policy with the example bucket policy in the README. For more information, see Prerequisites.

```
python3 liveanalytics_influxdb3_migration_client.py \
    --live-analytics-database-name <your LiveAnalytics database name> \
    --s3-bucket-name <your S3 bucket name>
```

Output: Timestream for LiveAnalytics database is transformed to line protocol and Ingested to InfluxDB 3 database.

Anomaly detection plugins

MAD-based anomaly detection

- **Trigger types**: Data write (real-time)
- **Use cases**: Real-time outlier detection for streaming data, sensor monitoring, quality control.
- **GitHub**: MAD Anomaly Detection Documentation

How it works: Uses Median Absolute Deviation (MAD) to establish a robust baseline for normal behavior. As new data arrives, it calculates how many MADs away from the median each point is. Points exceeding the threshold (k * MAD) are flagged as anomalies.

Key features:

- Real-time processing as data is written.
- Maintains in-memory sliding windows for efficiency.
- Count-based alerts (e.g., 5 consecutive anomalies).
- Duration-based alerts (e.g., anomaly for 2 minutes).
- Flip suppression to prevent alert fatigue from rapidly changing values.

Example usage:

```
# Detect temperature anomalies in real-time
influxdb3 create trigger \
  --database sensors \
  --plugin-filename "mad_check/mad_check_plugin.py" \
  --trigger-spec "all_tables" \
  --trigger-arguments
 'measurement=temperature_sensors,mad_thresholds="temp:2.5:20:5@humidity:3:30:2m",senders=slack
 \
```

```
  temp_anomaly_detector

# Threshold format: field:k_multiplier:window_size:trigger_condition
# temp:2.5:20:5 = temperature field, 2.5 MADs, 20-point window, alert after 5
 consecutive anomalies
# humidity:3:30:2m = humidity field, 3 MADs, 30-point window, alert after 2 minutes of
 anomaly
```

Output: Sends real-time notifications when anomalies are detected, including the field name, value, and duration.

Data transformation plugins

Basic transformation

- **Trigger types**: Scheduled, Data write
- **Use cases**: Data standardization, unit conversions, field name normalization, data cleaning.
- **GitHub**: Basic Transformation Documentation

How it works: Applies a chain of transformations to field names and values. Can process historical data in batches (scheduled) or transform data as it arrives (data write). Transformations are applied in the order specified, allowing complex data pipelines.

Key features:

- Field name transformations: snake_case, remove spaces, alphanumeric only.
- Unit conversions: Temperature, pressure, length, time units.
- Custom string replacements with regex support.
- Dry-run mode for testing without writing data.
- Batch processing for historical data.

Example usage:

```
# Transform temperature data from Celsius to Fahrenheit with field name standardization
influxdb3 create trigger \
  --database weather \
  --plugin-filename "basic_transformation/basic_transformation.py" \
  --trigger-spec "every:30m" \
```

```
  --trigger-arguments
 'measurement=raw_weather,window=1h,target_measurement=weather_fahrenheit,names_transformations
 Reading":"snake",values_transformations=temperature_reading:"convert_degC_to_degF"' \
  temp_converter

# Real-time field name cleaning for incoming sensor data
influxdb3 create trigger \
  --database iot \
  --plugin-filename "basic_transformation/basic_transformation.py" \
  --trigger-spec "all_tables" \
  --trigger-arguments
 'measurement=raw_sensors,target_measurement=clean_sensors,names_transformations=.*:"snake
 alnum_underscore_only collapse_underscore"' \
  sensor_cleaner
```

Output: Creates a new table with transformed data, preserving original timestamps and tags.

Downsampler

- **Trigger types**: Scheduled, HTTP
- **Use cases**: Data reduction, long-term storage optimization, creating summary statistics, performance improvement.
- **GitHub**: Downsampler Documentation

How it works: Aggregates high-resolution time-series data into lower-resolution summaries. For example, converts 1-second data into 1-hour averages. Each downsampled point includes metadata about the number of original points compressed and the time range covered.

Key features:

- Multiple aggregation functions: avg, sum, min, max, median, derivative.
- Field-specific aggregations (different functions for different fields).
- Metadata tracking (record_count, time_from, time_to).
- HTTP API for on-demand downsampling with backfill.
- Configurable batch sizes for large datasets.

Example usage:

```
# Downsample CPU metrics from 10-second to hourly resolution
```

```
influxdb3 create trigger \
  --database metrics \
  --plugin-filename "downsampler/downsampler.py" \
  --trigger-spec "every:1h" \
  --trigger-arguments
 'source_measurement=cpu_detailed,target_measurement=cpu_hourly,interval=1h,window=6h,calculati
 \
  cpu_downsampler

# HTTP endpoint for on-demand downsampling
curl -X POST http://localhost:8086/api/v3/engine/downsample \
  -H "Authorization: Bearer YOUR_TOKEN" \
  -d '{
    "source_measurement": "sensor_data",
    "target_measurement": "sensor_daily",
    "interval": "1d",
    "calculations": [["temperature", "avg"], ["humidity", "avg"], ["pressure", "max"]],
    "backfill_start": "2024-01-01T00:00:00Z",
    "backfill_end": "2024-12-31T23:59:59Z"
  }'
```

Output: Creates downsampled data with aggregated values plus metadata columns showing the number of points compressed and time range.

Monitoring and alerting plugins

State change monitor

- **Trigger types**: Scheduled, Data write
- **Use cases**: Status monitoring, equipment state tracking, process monitoring, change detection.
- **GitHub**: State Change Documentation

How it works: Tracks field value changes over time and alerts when the number of changes exceeds configured thresholds. Can detect both value changes (different values) and specific value conditions (equals a target value). Includes stability checks to prevent alerts from noisy signals.

Key features:

- Count-based change detection (e.g., five changes in ten minutes).
- Duration-based monitoring (e.g., status = "error" for five minutes).
- State change window for noise reduction.

- Multi-field monitoring with independent thresholds.
- Configurable stability requirements.

Example usage:

```
# Monitor equipment status changes
influxdb3 create trigger \
  --database factory \
  --plugin-filename "state_change/state_change_check_plugin.py" \
  --trigger-spec "every:5m" \
  --trigger-arguments
 'measurement=equipment,field_change_count="status:3.temperature:10",window=15m,state_change_wi
 $field changed $changes times in $window"' \
  equipment_monitor

# Real-time monitoring for specific state conditions
influxdb3 create trigger \
  --database systems \
  --plugin-filename "state_change/state_change_check_plugin.py" \
  --trigger-spec "all_tables" \
  --trigger-arguments
 'measurement=service_health,field_thresholds="status:down:5@health_score:0:10",senders=pagerdu
 \
  service_monitor
```

Output: Alerts include the field name, number of changes detected, time window, and relevant tag values.

System metrics collector

- **Trigger types**: Scheduled
- **Use cases**: Infrastructure monitoring, performance baselines, capacity planning, resource tracking.
- **GitHub**: System Metrics Documentation

How it works: Uses the psutil library to collect comprehensive system metrics from the host running InfluxDB. Collects CPU, memory, disk, and network statistics at configurable intervals. Each metric type can be enabled/disabled independently.

Key features:

- Per-core CPU statistics with load averages.
- Memory usage including swap and page faults.
- Disk I/O metrics with calculated IOPS and latency.
- Network interface statistics with error tracking.
- Configurable metric collection (enable/disable specific types).
- Automatic retry on collection failures.

Example usage:

```
# Collect all system metrics every 30 seconds
influxdb3 create trigger \
  --database monitoring \
  --plugin-filename "system_metrics/system_metrics.py" \
  --trigger-spec "every:30s" \
  --trigger-arguments 'hostname=db-
server-01,include_cpu=true,include_memory=true,include_disk=true,include_network=true'
 \
  system_monitor

# Focus on CPU and memory for application servers
influxdb3 create trigger \
  --database app_monitoring \
  --plugin-filename "system_metrics/system_metrics.py" \
  --trigger-spec "every:1m" \
  --trigger-arguments 'hostname=app-
server-01,include_cpu=true,include_memory=true,include_disk=false,include_network=false'
 \
  app_metrics
```

Output: Creates multiple tables (system_cpu, system_memory, system_disk_io, etc.) with detailed metrics for each subsystem.

Common configuration patterns

Using TOML configuration files

For complex configurations, use TOML files instead of inline arguments:

```
# anomaly_config.toml
```

```
measurement = "server_metrics"
field = "cpu_usage"
window = "1h"
detector_type = "IsolationForestAD"
contamination = 0.1
window_size = 20
output_table = "cpu_anomalies"
senders = "slack"
slack_webhook_url = "https://hooks.slack.com/services/YOUR/WEBHOOK"
notification_text = "Anomaly detected in $field: value=$value at $timestamp"
```

```
# Use TOML configuration
PLUGIN_DIR=~/.plugins influxdb3 create trigger \
  --database monitoring \
  --plugin-filename "adtk_anomaly/adtk_anomaly_detection_plugin.py" \
  --trigger-spec "every:10m" \
  --trigger-arguments "config_file_path=anomaly_config.toml" \
  cpu_anomaly_detector
```

Chaining plugins

Create data processing pipelines by chaining multiple plugins:

```
# Step 1: Transform raw data
influxdb3 create trigger \
  --database pipeline \
  --plugin-filename "basic_transformation/basic_transformation.py" \
  --trigger-spec "all_tables" \
  --trigger-arguments
 'measurement=raw_sensors,target_measurement=clean_sensors,names_transformations=.*:"snake"'
 \
  step1_transform

# Step 2: Downsample transformed data
influxdb3 create trigger \
  --database pipeline \
  --plugin-filename "downsampler/downsampler.py" \
  --trigger-spec "every:1h" \
  --trigger-arguments
 'source_measurement=clean_sensors,target_measurement=sensors_hourly,interval=1h,window=6h,calc
 \
  step2_downsample
```

```
# Step 3: Detect anomalies in downsampled data
influxdb3 create trigger \
  --database pipeline \
  --plugin-filename "mad_check/mad_check_plugin.py" \
  --trigger-spec "all_tables" \
  --trigger-arguments
 'measurement=sensors_hourly,mad_thresholds="value:3:20:5",senders=slack' \
  step3_anomaly
```

Best practices for plugins

- **Start conservative** – Begin with higher thresholds and longer windows, then adjust based on observed patterns.
- **Test in development** – Use dry-run modes and test databases before production deployment.
- **Monitor plugin performance** – Check execution times and resource usage in system tables.
- **Use appropriate trigger types** – Choose scheduled for batch processing, data write for real-time.
- **Configure notifications wisely** – Use severity levels and debounce logic to prevent alert fatigue.
- **Leverage model persistence** – For ML-based plugins, save trained models for consistency.
- **Document configurations** – Use descriptive trigger names and maintain configuration documentation.

Monitor plugin execution

To monitor plugin performance:

```
-- View plugin execution logs
SELECT
  event_time,
  trigger_name,
  log_level,
  log_text
FROM system.processing_engine_logs
WHERE trigger_name = 'your_trigger_name'
  AND time > now() - INTERVAL '1 hour'
ORDER BY event_time DESC;

-- Monitor plugin performance
SELECT
  trigger_name,
```

```
  COUNT(*) as executions,
  AVG(execution_time_ms) as avg_time_ms,
  MAX(execution_time_ms) as max_time_ms,
  SUM(CASE WHEN log_level = 'ERROR' THEN 1 ELSE 0 END) as error_count
FROM system.processing_engine_logs
WHERE time > now() - INTERVAL '24 hours'
GROUP BY trigger_name;

-- Check trigger status
SELECT * FROM system.processing_engine_triggers
WHERE database = 'your_database';
```

Troubleshoot common issues

The following table shows common issues and possible solutions.

Issue	Solution
Plugin not triggering	Verify trigger is enabled, check schedule/spec syntax
Missing notifications	Confirm Notifier plugin installed, check webhook URLs
High memory usage	Reduce window sizes, adjust batch processing intervals
Incorrect transformations	Use dry-run mode, verify field names and data types
Forecast inaccuracy	Increase training data window, adjust seasonality settings
Too many alerts	Increase trigger counts, add debounce duration, adjust thresholds

These certified plugins provide enterprise-ready functionality for common time-series data processing needs, eliminating the need for custom development while maintaining flexibility through comprehensive configuration options. Visit the GitHub repository for detailed documentation, examples, and updates.

Use custom plugins with the processing engine

Amazon Timestream for InfluxDB 3 lets you run your own Python plugins on the managed processing engine. With custom plugins, you host your plugin code in one or more plugin

repositories that you control – public or private – and the engine fetches and runs it in response to triggers (data writes, schedules, or HTTP requests). Use custom plugins to implement logic that is specific to your workload: custom data transformations, alerting, aggregation, and integrations with your own services.

You enable custom plugins by setting a plugin repository on a DB parameter group and applying that parameter group to your cluster. Custom plugins run on both Core and Enterprise editions.

How custom plugins work

You configure a plugin repository URL – and, for a private repository, a secret that holds an access token – as parameters on a DB parameter group. After your cluster uses that parameter group, you reference a plugin in a trigger with the `gh:` prefix. The engine resolves the path against your repository URL and fetches the raw file over HTTPS:

```
gh:<path-to-plugin-file>   #   <plugin-repository-url>/<path-to-plugin-file>
```

For example, with a plugin repository URL of `https://raw.githubusercontent.com/my-org/influxdb-plugins/main/`, a trigger created with `--path "gh:transforms/enrich.py"` fetches:

```
https://raw.githubusercontent.com/my-org/influxdb-plugins/main/transforms/enrich.py
```

> **Note**
>
> The `gh:` prefix supports single-file plugins only. Each custom plugin must be a self-contained `.py` file addressable by a stable path under your repository URL.

Prerequisites

- An Amazon Timestream for InfluxDB 3 cluster (Core or Enterprise), with custom plugins enabled for your account.
- A plugin repository reachable over HTTPS that serves raw plugin files (public or private).
- Permission to create and apply DB parameter groups (`timestream-influxdb:CreateDbParameterGroup`, `timestream-influxdb:UpdateDbCluster`).
- An InfluxDB 3 admin token for your cluster, to create and update triggers.

- (Private repositories only) A read-only repository access token, and permission to create a secret in AWS Secrets Manager.

Step 1: Create a parameter group with the plugin repository

Create a DB parameter group that sets the custom plugin repository. Set both parameters: the repository URL, and – for a private repository – the ARN of the Secrets Manager secret that holds your repository access token.

```
aws timestream-influxdb create-db-parameter-group \
  --name my-plugin-parameter-group \
  --description "InfluxDB 3 cluster with a custom plugin repository" \
  --region us-west-2 \
  --parameters '{
    "InfluxDBv3Core": {
      "pluginRepositoryUrl": "https://raw.githubusercontent.com/my-org/influxdb-
plugins/main/",
      "pluginRepositorySecretArn": "arn:aws:secretsmanager:us-
west-2:111122223333:secret:InfluxDB-RepoToken-my-plugins-AbCdEf"
    }
  }'
```

Set the parameters under `InfluxDBv3Core`, or `InfluxDBv3Enterprise` for an Enterprise cluster:

- `pluginRepositoryUrl` – The base HTTPS URL that serves your raw plugin files. It must use HTTPS with a valid host, and must not include embedded credentials, query strings, or fragments. The base can be a full repository path (recommended – it confines the cluster to that one repository) or a host-only URL if you need plugins from more than one repository (see the section called "Use multiple plugin repositories"). Include the branch (for example, `.../main/`) so plugin versions are predictable.

  The URL is normalized when you create the parameter group: a bare GitHub repository URL (`https://github.com/<owner>/<repo>`) is automatically rewritten to its raw-content form (`https://raw.githubusercontent.com/<owner>/<repo>`), and a trailing slash is removed (`.../main/` becomes `.../main`). GitHub `/tree/<branch>` and `/blob/...` web URLs are not rewritten – provide a raw-content base URL for those.
- `pluginRepositorySecretArn` – The ARN of the Secrets Manager secret that holds your repository access token. Required only for private repositories. The secret must be in the

same AWS account and AWS Region as the parameter group, and its name must begin with `InfluxDB-RepoToken-` (see the section called "(Optional) Create the repository access secret"). Omit this parameter for public repositories.

For a public repository, set only the URL:

```
aws timestream-influxdb create-db-parameter-group \
  --name my-plugin-parameter-group \
  --description "InfluxDB 3 cluster with a public custom plugin repository" \
  --region us-west-2 \
  --parameters '{
    "InfluxDBv3Core": {
      "pluginRepositoryUrl": "https://raw.githubusercontent.com/my-org/influxdb-
plugins/main/"
    }
  }'
```

> **Important**
>
> The plugin repository parameters are immutable on an existing parameter group – there is no `UpdateDbParameterGroup` operation. To change the repository URL or secret, create a new parameter group and apply it to the cluster with `update-db-cluster`.

(Optional) Create the repository access secret

If your plugin repository is private, create a Secrets Manager secret that holds a repository access token, then pass its ARN as `pluginRepositorySecretArn` in Step 1. Skip this section for public repositories.

The secret must meet all of the following requirements, which are validated when you create the parameter group:

- Its name begins with `InfluxDB-RepoToken-`. This is the only name pattern Timestream for InfluxDB is authorized to read; a secret with any other name is rejected.
- It is in the same AWS account as the caller and the same AWS Region as the parameter group.
- Its value is the access token itself, as a plain string (not JSON).

1. In your repository provider, create a fine-grained, read-only access token scoped to only the plugin repositories you reference. Fine-grained tokens (for example, a GitHub fine-grained personal access token) let you limit access to specific repositories with read-only contents permission and an expiration date – prefer them over classic or broad tokens. With a host-only base URL, the same token is used for every fetch from the configured host, so it must have read access to each private repository your triggers reference.
2. Store the token in Secrets Manager, with a name that begins with `InfluxDB-RepoToken-`:

   ```
   aws secretsmanager create-secret \
     --name InfluxDB-RepoToken-my-plugins \
     --secret-string "YOUR_REPOSITORY_ACCESS_TOKEN" \
     --region us-west-2
   ```

3. Copy the returned ARN and use it as the `pluginRepositorySecretArn` value in your parameter group.

By default, the secret is encrypted with the AWS managed key for Secrets Manager (`aws/secretsmanager`), which requires no additional configuration.

Use a customer managed AWS KMS key (CMK)

You can encrypt the repository token secret with a customer managed AWS KMS key. No changes to the key policy are required for the default case: Timestream for InfluxDB reads the secret using its service-linked role in your account, which already has scoped permission to decrypt `InfluxDB-RepoToken-*` secrets. Create the secret with the CMK and reference its ARN as usual:

```
aws secretsmanager create-secret \
  --name InfluxDB-RepoToken-my-plugins \
  --secret-string "YOUR_REPOSITORY_ACCESS_TOKEN" \
  --kms-key-id arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab \
  --region us-west-2
```

> **Note**
>
> This works with a key that uses the default AWS KMS key policy (which delegates access to IAM identities in your account). If you replace the default key policy with a custom one that removes IAM delegation, you must ensure the Timestream for InfluxDB service-linked role

(`AWSServiceRoleForTimestreamInfluxDB`) can still call `kms:Decrypt` on the key – otherwise plugin fetches fail with a AWS KMS access-denied error.

Step 2: Apply the parameter group to your cluster

Associate the parameter group with a cluster, either when you create the cluster or by updating an existing cluster. Applying a parameter group runs a cluster maintenance action, so plan for a maintenance window.

New cluster:

```
aws timestream-influxdb create-db-cluster \
  --name my-cluster \
  --db-instance-type db.influx.large \
  --db-parameter-group-identifier my-plugin-parameter-group \
  --vpc-subnet-ids subnet-0123456789abcdef0 subnet-0abcdef1234567890 \
  --vpc-security-group-ids sg-0123456789abcdef0 \
  --region us-west-2
```

Existing cluster:

```
aws timestream-influxdb update-db-cluster \
  --db-cluster-id my-cluster-id \
  --db-parameter-group-identifier my-plugin-parameter-group \
  --region us-west-2
```

When the cluster returns to the `AVAILABLE` state, the custom plugin repository is active.

Step 3: Create a trigger that runs your custom plugin

Create triggers against your cluster's InfluxDB endpoint using the `influxdb3` CLI (or the HTTP API), and reference your custom plugin with the `gh:` prefix. The path after `gh:` is resolved against the `pluginRepositoryUrl` you configured in Step 1.

Trigger on writes to a table:

```
influxdb3 create trigger \
  --trigger-spec "table:sensor_data" \
  --path "gh:transforms/enrich.py" \
```

```
  --database DATABASE_NAME \
  --token YOUR_ADMIN_TOKEN \
  enrich_sensor_data
```

Trigger on a schedule:

```
influxdb3 create trigger \
  --trigger-spec "every:5m" \
  --path "gh:metrics/custom_system.py" \
  --database DATABASE_NAME \
  --token YOUR_ADMIN_TOKEN \
  custom_metrics
```

Trigger on an HTTP request:

```
influxdb3 create trigger \
  --trigger-spec "request:ingest" \
  --path "gh:alerting/threshold.py" \
  --database DATABASE_NAME \
  --token YOUR_ADMIN_TOKEN \
  threshold_endpoint
```

You can pass configuration to a custom plugin with `--trigger-arguments`, and control error handling and concurrency exactly as you do for other processing engine plugins. For the full command reference, see influxdb3 create trigger in the InfluxDB 3 documentation; for Amazon Timestream trigger configuration, see Configure triggers.

Use multiple plugin repositories

A parameter group registers one `pluginRepositoryUrl`, but if you set that base to the raw-content host only (no owner or repository), each trigger can select a different repository in its `gh:` path – for example, the public InfluxData repository and your own repository on the same cluster. The engine appends the trigger path to the base URL, so different triggers can load from different repositories.

Set the repository URL to the host with no owner or repository path:

```
{
  "InfluxDBv3Core": {
    "pluginRepositoryUrl": "https://raw.githubusercontent.com"
  }
```

```
}
```

Then give each trigger the full owner/repository/branch/path after `gh:`:

```
# Plugin from the public InfluxData repository
influxdb3 create trigger \
  --trigger-spec "every:5m" \
  --path "gh:influxdata/influxdb3_plugins/main/influxdata/signal_generator/
signal_generator.py" \
  --database DATABASE_NAME --token YOUR_ADMIN_TOKEN influxdata_signal

# Plugin from your own repository
influxdb3 create trigger \
  --trigger-spec "table:sensor_data" \
  --path "gh:my-org/my-plugins/main/transforms/enrich.py" \
  --database DATABASE_NAME --token YOUR_ADMIN_TOKEN my_enrich
```

Both resolve to `https://raw.githubusercontent.com/<the gh: path>`.

- Include the **branch** (for example, `main`) in every `gh:` path – the segment after the repository name is required.
- This applies to **public** repositories. For a private repository, also set `pluginRepositorySecretArn`. The token in that secret is used for every fetch from the configured host, so it must have read access to each private repository you reference.

> **Important**
>
> A host-only base lets the cluster load plugin code from *any* repository the host can serve – not only the ones you intend. Use it only when you genuinely need multiple repositories. When you use a single repository, pin the base to `https://raw.githubusercontent.com/<owner>/<repo>/<branch>/` so the cluster is confined to that one repository. Always point production clusters at reviewed, trusted repositories.

Update custom plugins

To update a plugin after you push changes to your repository, use the `influxdb3 update trigger` command against your cluster endpoint. This re-fetches the plugin and applies the change to the running trigger without recreating it – no cluster reboot is required:

```
influxdb3 update trigger \
  --database DATABASE_NAME \
  --token YOUR_ADMIN_TOKEN \
  --trigger-name enrich_sensor_data \
  --path "gh:transforms/enrich.py"
```

To change the plugin *repository* itself (URL or secret), create a new parameter group with the new values and apply it to the cluster with `update-db-cluster`. The repository parameters are immutable on an existing parameter group.

Security considerations

Custom plugins execute code that you supply inside your database engine. Treat plugin code as you would any code with access to your data.

- **Only use repositories you trust and control.** Custom plugins run within your cluster with access to your database.
- **Use least-privilege, fine-grained repository tokens.** Scope the token to read-only access to only the plugin repositories your triggers reference, set an expiration, and rotate it periodically.
- **Protect the secret.** Restrict access to the Secrets Manager secret with IAM. Optionally encrypt it with a customer managed AWS KMS key.
- **Validate inputs in HTTP-request plugins.** Treat request bodies, headers, and query parameters as untrusted input.
- **Pin the repository base URL.** A host-only base URL lets the cluster load plugin code from any repository the host can serve – not only the ones you intend. When you use a single repository, pin the base to `https://raw.githubusercontent.com/<owner>/<repo>/<branch>/` so the cluster is confined to that one repository.
- **Use a vetted repository for production.** Point production clusters at a repository that contains only reviewed, approved plugin versions, and use change control for updates.

Restrict the secret and CMK to Timestream for InfluxDB (resource policies)

By default, access to the repository-token secret and its encryption key is governed by IAM in your account. To tightly scope access so that only Timestream for InfluxDB – and no other principal in your account – can read the token secret and decrypt it, attach the following resource policies. Both grant the Timestream for InfluxDB service-linked role (`AWSServiceRoleForTimestreamInfluxDB`, created automatically in your account) exactly the

two permissions it needs, and nothing more. Replace `<account-id>` and `<region>` with your values.

Secret resource policy (Secrets Manager)

Grants only the service-linked role `secretsmanager:GetSecretValue` on the secret:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowTimestreamInfluxDBReadRepoToken",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::<account-id>:role/aws-service-role/timestream-
influxdb.amazonaws.com/AWSServiceRoleForTimestreamInfluxDB"
      },
      "Action": "secretsmanager:GetSecretValue",
      "Resource": "*"
    }
  ]
}
```

Apply it (`"Resource": "*"` refers to the secret the policy is attached to):

```
aws secretsmanager put-resource-policy \
  --secret-id InfluxDB-RepoToken-my-plugins \
  --resource-policy file://secret-resource-policy.json \
  --region <region>
```

CMK key policy (AWS KMS)

If the secret is encrypted with a customer managed key, attach a key policy that grants only the service-linked role `kms:Decrypt` / `kms:DescribeKey`, and only when the key is used through Secrets Manager (`kms:ViaService`). Keep the `EnableIAMRoot` statement so you don't lose administrative control of the key:

```
{
  "Version": "2012-10-17",
  "Id": "timestream-repo-token-key-policy",
  "Statement": [
    {
```

```
      "Sid": "EnableIAMRoot",
      "Effect": "Allow",
      "Principal": { "AWS": "arn:aws:iam::<account-id>:root" },
      "Action": "kms:*",
      "Resource": "*"
    },
    {
      "Sid": "AllowTimestreamInfluxDBDecryptViaSecretsManager",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::<account-id>:role/aws-service-role/timestream-
influxdb.amazonaws.com/AWSServiceRoleForTimestreamInfluxDB"
      },
      "Action": [
        "kms:Decrypt",
        "kms:DescribeKey"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "secretsmanager.<region>.amazonaws.com"
        }
      }
    }
  ]
}
```

Apply it:

```
aws kms put-key-policy \
  --key-id <cmk-key-id> \
  --policy-name default \
  --policy file://cmk-key-policy.json \
  --region <region>
```

Note

- Keep the `EnableIAMRoot` statement – removing it can lock you out of the key.
- These resource-side grants scope access independently of the service-linked role's own identity policy, so they are the reliable way to restrict the secret and key to only Timestream for InfluxDB.

- To scope the key even more tightly to this one secret, add a `"kms:EncryptionContext:SecretARN": "<secret-arn>"` entry to the `StringEquals` condition.

Python dependencies

Custom plugins run in a managed Python environment that includes the Python standard library and a set of pre-installed, Amazon-vetted libraries (for example, `numpy`, `pandas`, and `httpx`). To keep the environment secure and consistent, the Python package manager is **disabled** – you cannot install additional third-party packages from within a plugin or on the cluster. Write your custom plugins to use only the standard library and the pre-installed libraries.

If a plugin imports a package that is not available in the environment, the trigger fails when it runs. Check `system.processing_engine_logs` for the import error.

Limitations

- Only single-file plugins are supported from a custom repository.
- The plugin repository must serve raw file bytes over HTTPS, with no embedded credentials, query strings, or fragments in the URL.
- The repository access secret must be in the same AWS account and AWS Region as the parameter group and named `InfluxDB-RepoToken-*`.
- The plugin repository parameters on a parameter group are immutable. To change the repository, create a new parameter group and apply it with `update-db-cluster`.

Monitor custom plugin execution

Custom plugins are monitored the same way as certified plugins. Query the processing engine system tables to confirm a plugin loaded and to troubleshoot failures:

```
-- View processing engine logs for a trigger
SELECT * FROM system.processing_engine_logs
WHERE trigger_name = 'enrich_sensor_data'
  AND time > now() - INTERVAL '1 hour'
ORDER BY time DESC;
```

```
-- Check trigger status
SELECT * FROM system.processing_engine_triggers
WHERE database = 'DATABASE_NAME';
```

Troubleshoot custom plugins

Issue	Possible cause and resolution
`CreateDbParameterGroup` rejects the secret ARN	The secret must be in the same AWS account and AWS Region as the parameter group and named `InfluxDB-RepoToken-*`. Verify all three.
Invalid `pluginRepositoryUrl`	The URL must be HTTPS with a valid host, and must not contain credentials, query strings, or fragments. A host-only base URL (no repository path) is accepted.
Plugin fetch fails with 404 Not Found	The plugin path doesn't exist at the resolved URL. Verify the `gh:` path and that `pluginRepositoryUrl` includes the correct branch. Note that GitHub `/tree/<branch>` and `/blob/...` URLs are not supported – use a raw-content base URL.
Plugin fetch fails with 401/403 (private repository)	The access token is missing, expired, or lacks read access. Verify the token in the `InfluxDB-RepoToken-*` secret and its scope.
AWS KMS access-denied reading the secret	You replaced the default key policy on your CMK. Ensure the Timestream for InfluxDB service-linked role can `kms:Decrypt` the key.
Updated a plugin but the trigger runs the old version	Run `influxdb3 update trigger` to re-fetch the updated plugin for that trigger.
Plugin fails with a Python import error	The plugin imports a package that is not in the managed Python environment. The package manager is disabled; use only the standard library and pre-installed libraries. Check `system.processing_engine_logs`.

Encrypting resources with customer managed keys

Amazon Timestream for InfluxDB 3 encrypts your data at rest by default using AWS owned encryption keys. You can optionally specify a customer managed key (CMK) when creating a new cluster to control data encryption with your own AWS Key Management Service (AWS KMS) key.

Overview

When you specify a customer managed key during cluster creation, Amazon Timestream for InfluxDB 3 uses your key to encrypt the Amazon S3 objects that store your database data.

Using a customer managed key gives you full control over the encryption key lifecycle, including the ability to create, rotate, disable, and define access policies. You can also audit key usage through AWS CloudTrail.

> **Important**
>
> You can only specify a customer managed key during cluster creation. You cannot change the encryption key after a cluster has been created.

Supported key types

Amazon Timestream for InfluxDB 3 requires an AWS KMS symmetric encryption key (`SYMMETRIC_DEFAULT`). The key must meet these requirements:

- Key type: Symmetric (`SYMMETRIC_DEFAULT`)
- Key usage: Encrypt and decrypt
- Key origin: AWS KMS (customer managed)
- Key Region: Same AWS Region as the cluster
- Multi-Region keys: Supported (the primary or replica key must be in the same Region as the cluster)

Creating a cluster with a customer managed key

Using the AWS CLI

To create an InfluxDB 3 cluster with a customer managed key, specify the `--kms-key-id` parameter with the ARN of your AWS KMS key.

```
aws timestream-influxdb create-db-cluster \
    --name my-influxdb3-cluster \
    --db-instance-type db.influx.medium \
    --vpc-subnet-ids subnet-0123456789abcdef0 subnet-0123456789abcdef1 \
    --vpc-security-group-ids sg-0123456789abcdef0 \
    --db-parameter-group-identifier InfluxDBV3Core \
    --kms-key-id arn:aws:kms:us-
east-1:123456789012:key/12345678-1234-1234-1234-123456789012
```

Using the

1. Open the Amazon Timestream for InfluxDB console.
2. Choose **Create database**.
3. In the **Encryption** section, select **Customer managed key**.
4. Choose your AWS KMS key from the dropdown list, or enter the key ARN.
5. Complete the remaining configuration and choose **Create**.

Setting up the KMS key policy

Before creating a cluster with a customer managed key, you must configure the key policy to grant Amazon Timestream for InfluxDB 3 the necessary permissions. The key policy requires two statements: one to allow the calling role to create grants through the service, and one to allow the role to describe the key.

Add the following statements to your key policy, replacing *`account-id`*, *`caller-role`*, and *`region`* with your values:

```
{
    "Sid": "Allow Timestream InfluxDB to use the key for resource allocations",
    "Effect": "Allow",
    "Principal": {
        "AWS": "arn:aws:iam::account-id:role/caller-role"
```

```
    },
    "Action": "kms:CreateGrant",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "kms:ViaService": "timestream-influxdb.region.amazonaws.com"
        },
        "ForAllValues:StringEquals": {
            "kms:GrantOperations": [
                "Decrypt",
                "Encrypt",
                "GenerateDataKey",
                "GenerateDataKeyWithoutPlaintext",
                "ReEncryptFrom",
                "ReEncryptTo",
                "CreateGrant",
                "DescribeKey"
            ]
        },
        "Bool": {
            "kms:GrantIsForAWSResource": "true"
        }
    }
},
{
    "Sid": "Allow Timestream InfluxDB to describe the key for resource allocations",
    "Effect": "Allow",
    "Principal": {
        "AWS": "arn:aws:iam::account-id:role/caller-role"
    },
    "Action": "kms:DescribeKey",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "kms:ViaService": "timestream-influxdb.region.amazonaws.com"
        }
    }
}
```

The first statement allows the caller's IAM role to create grants through the Amazon Timestream for InfluxDB 3 service, scoped to specific grant operations. The `kms:ViaService` condition ensures this permission can only be exercised through the service, and `kms:GrantIsForAWSResource` ensures grants are only created for AWS resources. Though

InfluxDB 3 uses Amazon S3 as the data store, the grant operations are needed because the service also uses Amazon EBS to store the InfluxDB logs.

The second statement allows the caller's IAM role to describe the key through the service.

Viewing the encryption key

To view the AWS KMS key used by a cluster, use the `GetDbCluster` API. The response includes the `kmsKeyId` field if a customer managed key was specified during creation.

```
aws timestream-influxdb get-db-cluster \
    --db-cluster-identifier my-influxdb3-cluster
```

If no customer managed key was specified, the `kmsKeyId` field is not present in the response, indicating that the cluster uses the default AWS owned key.

Monitoring key usage with CloudTrail

When you use a customer managed key, AWS CloudTrail records all AWS KMS API calls made by Amazon Timestream for InfluxDB 3. You can use these logs to audit key usage and monitor access patterns.

Event	When it occurs
`CreateGrant`	When provisioning Amazon S3 bucket and service managed backups
`GenerateDataKey`	When writing data to Amazon S3 (encrypting new objects)
`Decrypt`	When reading data from Amazon S3 (decrypting objects)

The `invokedBy` field in these events shows `timestream-influxdb.`*`region`*`.amazonaws.com`, identifying the service that initiated the request.

Key revocation and deletion

If you disable or delete a customer managed key, Amazon Timestream for InfluxDB 3 loses access to the encrypted data. This results in the following behavior:

- The cluster becomes unavailable.
- Read and write operations fail.
- The cluster remains in a degraded state until key access is restored.

If you re-enable the key, the cluster automatically recovers and becomes operational again.

> **Warning**
>
> If you permanently delete a customer managed key (after the AWS KMS waiting period), the encrypted data is permanently unrecoverable. Always ensure that you no longer need access to the data before scheduling key deletion.

Service-managed backups: For InfluxDB 3, service-managed backups are encrypted with the same customer managed key via an AWS Backup vault. If the key is deleted or revoked, existing backups cannot be restored.

Migrating existing clusters to use a customer managed key

Because you cannot change the encryption key after cluster creation, migrating an existing cluster to use a customer managed key requires:

1. Create a new cluster with the `--kms-key-id` parameter.
2. Migrate data from the existing cluster to the new cluster using the standard write APIs, or use customer-managed snapshots if available.
3. Update your application connection settings to point to the new cluster.
4. Delete the old cluster when migration is complete.

Limitations

- Customer managed keys can only be specified during cluster creation—not updated afterward.
- The AWS KMS key must be in the same AWS Region and account as the cluster.

- Only symmetric encryption keys (`SYMMETRIC_DEFAULT`) are supported.
- Root volumes (operating system) are not encrypted with the customer managed key—they continue to use service-managed encryption.
- In-place restore (`REPLACE_EXISTING` mode) is not supported for CMK-encrypted clusters. To restore a CMK-encrypted cluster, use `NEW_RESOURCE` mode to create a new cluster from the backup.

Pricing

There is no additional Amazon Timestream for InfluxDB 3 charge for using customer managed keys. Standard AWS KMS pricing applies for your key and the API calls made by the service. For details, see AWS KMS pricing.

Amazon Timestream for InfluxDB 3 frequently asked questions

Amazon Timestream for InfluxDB 3 is a fully managed time-series database service on AWS built on Apache Arrow, Apache DataFusion, and Apache Parquet. It supports SQL and InfluxQL queries, line protocol writes, and is available in Core (single-node, real-time monitoring) and Enterprise (multi-node, compaction, high availability) editions. This section answers the most common questions organized by topic.

General

What Amazon Timestream for InfluxDB 3 is, how it differs from v2, Core vs Enterprise editions, regional availability, and event notifications.

Clusters and instances

Instance types from db.influx.medium to db.influx.24xlarge, creating clusters, multi-node Enterprise deployments, scaling, cluster endpoints, and maintenance windows.

Writing and querying data

Write APIs (v3, v2, v1 compatibility), SQL and InfluxQL query support, token-based authentication, line protocol format, and data plane administration.

Configuration and parameter groups

Creating and managing parameter groups, immutable parameters, cloning best practices, and instance-size tuning guidelines.

Schema design

Tags vs fields, high-cardinality handling, and schema design best practices for InfluxDB 3.

Processing engine

Running Python plugins inside your cluster with write, schedule, and request triggers.

Custom plugins

Running your own Python plugins from public and private repositories: parameter group setup, repository access secrets, multi-repository configurations, and troubleshooting.

Storage and billing

Decoupled compute/storage architecture on Amazon S3, billing components, automatic backups, and customer-managed backup and restore.

Security and access

VPC deployment, AWS Identity and Access Management access control, customer managed encryption keys (CMK), and managed security policies.

Topics

- General FAQ for Amazon Timestream for InfluxDB 3
- Clusters and instances FAQ for Amazon Timestream for InfluxDB 3
- Writing and querying data FAQ for Amazon Timestream for InfluxDB 3
- Configuration and parameter groups FAQ for Amazon Timestream for InfluxDB 3
- Schema design FAQ for Amazon Timestream for InfluxDB 3
- Processing engine FAQ for Amazon Timestream for InfluxDB 3
- Custom plugins FAQ for Amazon Timestream for InfluxDB 3
- Storage and billing FAQ for Amazon Timestream for InfluxDB 3
- Security and access FAQ for Amazon Timestream for InfluxDB 3

General FAQ for Amazon Timestream for InfluxDB 3

Common questions about Amazon Timestream for InfluxDB 3, its architecture, editions, and regional availability. For a full overview, see the section called “What is Timestream for InfluxDB 3?”.

What is Amazon Timestream for InfluxDB 3?

Amazon Timestream for InfluxDB 3 is a managed time-series database service that runs InfluxDB 3 on AWS. It is designed for large-scale time-series analytics using open-source APIs. The service handles provisioning, patching, backups, and software updates so you can focus on your applications.

How is InfluxDB 3 different from InfluxDB v2?

InfluxDB 3 is a complete architectural redesign. It replaces the TSM (Time-Structured Merge tree) storage engine with Apache Arrow for in-memory processing, Apache DataFusion for query execution, and Apache Parquet for columnar storage on Amazon S3. This enables better performance for high-cardinality data and more efficient scaling for analytical workloads.

What is the difference between Core and Enterprise editions?

Core is a cost-effective, single-node edition optimized for real-time monitoring of recent data, typically around 3 days. It does not include compaction or multi-node deployments, which means query performance degrades as data ages. Core is best suited for dashboards, real-time alerting, and development or proof-of-concept workloads.

Enterprise is designed for production workloads that require high availability and long-term data retention. Key differences include:

- **Compaction** – Enterprise includes compaction capabilities that maintain query performance over time by optimizing Parquet file organization. Without compaction, Core accumulates small files that slow down queries on older data.
- **Multi-node clusters** – Enterprise supports multi-node deployments across multiple Availability Zones with dedicated ingest, query, and compactor nodes. Core is limited to a single node.
- **Read replicas** – Enterprise supports query-only nodes to scale read-heavy workloads independently from ingest.
- **Historical queries** – Enterprise includes single series indexing for optimized long-term data analysis, making it suitable for months or years of data retention.

Choose Core for near real-time monitoring where cost is a priority. Choose Enterprise when you need high availability, compaction, or long-term data analysis.

Core vs Enterprise Feature Comparison

Feature	Core	Enterprise
Deployment	Single-node	Multi-node across AZs
Compaction	Not included	Included
Read replicas	Not available	Query-only node scaling
Historical queries	Limited (~3 days)	Months/years with indexing
High availability	Single node	Multi-AZ
Best for	Dev/test, real-time dashboards	Production, long-term analytics

Which AWS Regions support Amazon Timestream for InfluxDB 3?

For the current list of supported AWS Regions and endpoints, see Amazon Timestream endpoints and quotas in the AWS General Reference.

Does Amazon Timestream for InfluxDB 3 support event notifications?

Yes. Amazon Timestream for InfluxDB publishes events to Amazon EventBridge when clusters undergo state changes, including creation, scaling, parameter group updates, and maintenance. You can create EventBridge rules to route events to targets such as Lambda, Amazon SNS, Amazon SQS, or Amazon CloudWatch Logs. Events use source `aws.timestream-influxdb`. For details, see the section called “Amazon EventBridge events”.

Is there an additional charge for event notifications?

No. Publishing events to Amazon EventBridge is included at no additional Amazon Timestream for InfluxDB charge. Standard Amazon EventBridge pricing applies for rule evaluation and target delivery, and downstream target costs (such as Lambda invocations or Amazon SNS deliveries) follow their respective service pricing.

In which Regions are event notifications available?

Event notifications are available in all AWS Regions where Amazon Timestream for InfluxDB is available. Events are published in the same Region as your database resource.

Do I need to enable event notifications?

No. Events are published automatically for all Amazon Timestream for InfluxDB resources. You do not need to opt in or configure anything on the database side. To receive events, create an Amazon EventBridge rule that matches the events you want and routes them to a target.

What types of operations generate events?

Events are generated for creation, deletion, compute scaling, storage scaling, port changes, parameter group updates, log delivery changes, maintenance window updates, reboots, node addition/removal, engine type conversion, and Multi-AZ/Single-AZ changes. Both successful completions and failures emit events.

Can I filter events by cluster name or event type?

Yes. Amazon EventBridge supports content-based filtering on any event field. You can filter by `detail.SourceIdentifier` (cluster name), `detail.EventCategories` (creation, notification, maintenance, failure), `detail.EventID` (specific event type), or `detail-type` (instance vs. cluster events).

How quickly are events delivered after an operation completes?

Amazon EventBridge provides at-least-once delivery with a 24-hour retry window. Events are emitted only when a workflow reaches a terminal state (success or failure), not when the operation starts. Under normal conditions events are delivered shortly after completion, but delivery timing depends on target availability and is not guaranteed within a specific timeframe.

Can I send events to another AWS account?

Yes. You can configure Amazon EventBridge rules to forward events to an event bus in another account for centralized observability. Configure the target event bus to accept events from your source account.

Clusters and instances FAQ for Amazon Timestream for InfluxDB 3

Questions about creating, configuring, and scaling Amazon Timestream for InfluxDB 3 clusters and instance types. For detailed instructions, see the section called “Manage InfluxDB 3 Clusters” and the section called “Configuring an InfluxDB 3 cluster”.

What instance types are available?

Amazon Timestream for InfluxDB 3 offers instance types ranging from `db.influx.medium` (1 vCPU, 8 GiB) to `db.influx.24xlarge` (96 vCPUs, 768 GiB). All instances use the `db.influxIOIncluded` class, which bundles I/O costs into compute pricing for predictable billing.

How do I create an InfluxDB 3 cluster?

You can create a cluster using the AWS Management Console, the AWS CLI, or the Amazon Timestream API. At minimum, you need to specify a cluster name, instance type, VPC subnet IDs, and security group IDs. For example, using the AWS CLI:

```
aws timestream-influxdb create-db-cluster \
  --name "my-cluster" \
  --db-instance-type db.influx.2xlarge \
  --vpc-subnet-ids subnet-12345abc subnet-67890def \
  --vpc-security-group-ids sg-12345abc
```

How do multi-node Enterprise clusters work?

Enterprise clusters support up to 15 nodes distributed across multiple Availability Zones, with dedicated writer/reader (up to 4), reader-only (up to 13), and compactor roles. AWS manages the cluster topology and node roles. Nodes communicate internally and use DNS-based traffic distribution for client connections. For details, see the section called “Scaling a cluster” and the InfluxDB 3 Enterprise clustering documentation.

How do I scale my cluster?

You can scale vertically by changing the instance type, or scale horizontally (Enterprise only) by adding writer/reader nodes (up to 4) or reader-only nodes (up to 13). To scale horizontally, create a new parameter group with your desired node configuration and apply it to your cluster. Use the `update-db-cluster` command or the AWS Management Console to apply changes. For more information, see the section called “Scaling a cluster”.

How do I upgrade from Core to Enterprise edition?

You can upgrade an existing Core cluster to Enterprise through the AWS Management Console. If this is the first Enterprise activation on your AWS account, the upgrade must be done via the console. After that initial activation, subsequent upgrades can also be performed using the AWS CLI or API. Note that this is a one-way operation—you cannot downgrade from Enterprise to

Core. After the upgrade, your existing data will be gradually compacted in the background while the cluster remains usable. For details, see the section called "Upgrade Core to Enterprise".

What should I expect after upgrading from Core to Enterprise regarding compaction?

Because Core edition does not include a compactor, all existing data must be compacted after the upgrade to Enterprise. How this affects your cluster depends on your configuration:

- **Single-node clusters**: The compactor shares the same node as the writer and reader. You will likely need additional capacity (a larger instance type) to allow the compactor to process all existing data without impacting your running workload. Consider scaling up before or immediately after the upgrade, and scaling back down once compaction catches up.
- **3-node clusters**: A dedicated compactor node is provisioned, but the full performance benefits of Enterprise (such as optimized query performance from compacted data) will not be realized until the compactor finishes processing all pre-existing data. The time required depends on the volume of data in the database, the current workload, and the instance size and configuration.

To speed up compaction, you can increase `compaction-max-num-files-per-plan` and reduce `compaction-check-interval` in your parameter group. Scaling up the instance type also gives the compactor more CPU and memory to work with. For details, see the section called "Category 4: Compaction" and the section called "Upgrade Core to Enterprise".

Why is my cluster creation failing?

Common causes include insufficient IAM permissions, invalid VPC subnet or security group IDs, or missing the required Amazon S3 VPC endpoint for private clusters. Verify that your IAM role has the `timestream-influxdb:CreateDbCluster` permission, that your subnets are in different Availability Zones, and that your security groups allow the required inbound ports. Check the AWS CloudTrail logs for the specific error message.

What are cluster endpoints and how do they work?

Multi-node Enterprise clusters provide two types of cluster endpoints: a **read/write endpoint** that routes traffic to writer/reader nodes, and a **read-only endpoint** that routes traffic to all nodes capable of read operations (only available when the cluster includes reader-only nodes). There are also node-specific endpoints for direct access to individual nodes, but these are not recommended for production use as they do not provide automatic failover. For more information, see the section called "Scaling a cluster".

What happens during a scaling operation?

When you scale a cluster by applying a new parameter group, the cluster restarts to apply the new node configuration. For multi-node clusters using cluster endpoints, traffic is automatically redistributed to available nodes during the update process. Nodes are distributed across multiple Availability Zones for high availability. You can monitor the scaling progress through the cluster status, which shows `UPDATING` during the operation. For details, see the section called "Scaling a cluster".

What is the dedicated compactor and when is it required?

The dedicated compactor is a node that handles background data optimization tasks such as compacting Parquet files in Amazon S3. It is required for Enterprise clusters with 3 or more nodes (`dedicatedCompactor: true`). By running compaction on a separate node, write and read performance on your other nodes is not impacted by background optimization. Single-node Enterprise clusters handle compaction on the same node alongside read and write operations.

How do maintenance windows work?

Every Timestream for InfluxDB resource has a weekly maintenance window during which routine maintenance such as OS patching and engine updates is performed. By default, the service manages the maintenance window automatically. You can specify a preferred window using the format `ddd:HH:MM-ddd:HH:MM` (for example, `Sun:02:00-Sun:04:00`) with a minimum duration of 2 hours and a maximum of 24 hours. For details, see the section called "Maintenance windows".

Can I set a maintenance window in my local timezone?

Yes. You can specify a timezone for your maintenance window using IANA timezone identifiers such as `America/New_York`, `Europe/London`, or `Asia/Tokyo`. The `timezone` field is required when configuring a maintenance schedule. The system handles Daylight Saving Time transitions automatically. For the full list of supported timezones, see the section called "Maintenance windows".

Writing and querying data FAQ for Amazon Timestream for InfluxDB 3

Questions about writing data to and querying data from Amazon Timestream for InfluxDB 3, including supported APIs, query languages, and authentication. For complete guides, see the

section called “Writing data to your InfluxDB 3 cluster” and the section called “Querying data from InfluxDB 3”.

How do I write data to InfluxDB 3?

InfluxDB 3 supports multiple write methods: the native v3 write API, the InfluxDB v2 compatibility API, and the InfluxDB v1 compatibility API. All methods accept line protocol format. Official client libraries are available for Python, Go, Java, JavaScript, and C#. For detailed write instructions, see the InfluxDB 3 Enterprise write data guide.

What query languages does InfluxDB 3 support?

InfluxDB 3 supports SQL and InfluxQL through the v3 HTTP query API. SQL is the primary query language, powered by the Apache DataFusion engine. InfluxQL is supported for backward compatibility. You can also query using the `influxdb3` CLI tool. For query syntax and examples, see the InfluxDB 3 Enterprise query data guide.

Example SQL query:

```
SELECT time, temperature, location
FROM weather
WHERE time > now() - INTERVAL '1 hour'
ORDER BY time DESC
LIMIT 100
```

How does authentication work for queries?

InfluxDB 3 uses token-based authentication. When you first set up your cluster, you create a master user account and password. In the Enterprise edition, you can create multiple tokens with different access levels:

- **Admin tokens** – Grant full access to all server actions, including CLI commands and API endpoints. The first admin token created is the operator token, which has full administrative privileges.
- **Resource tokens** – Grant granular permissions on specific resources such as individual databases and tables. You can assign read-only, write-only, or read-write access per database, enabling fine-grained access control for different applications and users.

Tokens work with the v3 HTTP query API, the `influxdb3` CLI, and the v1/v2 compatibility APIs. For more details, see the InfluxDB 3 Enterprise token management documentation.

Can I manage databases and tables through the data plane?

Yes. In InfluxDB 3 Enterprise, you can create, list, and delete databases and tables directly using the `influxdb3` CLI or the HTTP API. You can also manage Last Value Caches, Distinct Value Caches, and custom file indexes for query optimization. These data plane operations let you administer your cluster without using the AWS Management Console. For the full list of administrative operations, see the [InfluxDB 3 Enterprise administration guide](#).

Why is my query returning a 401 Unauthorized error?

A 401 error means your authentication token is missing, expired, or does not have the required permissions. Verify that you are passing the token in the `Authorization: Bearer` header (or via the `INFLUXDB3_AUTH_TOKEN` environment variable for the CLI). If using resource tokens, confirm the token has read permission on the target database. Regenerate the token if it may have been revoked.

Why are my queries slow?

Common causes include scanning too many Parquet files (check `query-file-limit`), insufficient DataFusion threads for your instance size, or a cold Parquet memory cache after a restart. For Enterprise clusters, verify that compaction is running — without compaction, small files accumulate and degrade query performance. See [the section called "Detailed Parameter Reference"](#) for tuning guidance.

What is line protocol?

Line protocol is a text-based format for writing time-series data to InfluxDB. Each line represents a single data point with a measurement name, optional tag set, field set, and optional timestamp. For the full specification, see the [InfluxDB 3 line protocol reference](#).

Example line protocol:

```
weather,location=us-east,station=A1 temperature=82.0,humidity=71.2
 1465839830100400200
weather,location=us-west,station=B2 temperature=68.5,humidity=45.8
 1465839830100400200
```

Configuration and parameter groups FAQ for Amazon Timestream for InfluxDB 3

Questions about configuring Amazon Timestream for InfluxDB 3 clusters using parameter groups. For the full parameter reference, see the section called "Parameter Groups for DB Clusters in Amazon Timestream".

How can I check the current running configuration of my cluster?

You can verify the effective parameter values on a running cluster by querying the `_internal` database's `nodes` table:

```
SELECT * FROM _internal.nodes
```

This returns the active configuration for each node in your cluster. Use this to verify parameter values before and after applying a new parameter group.

What are parameter groups?

Parameter groups contain engine configuration values that control how your InfluxDB 3 cluster operates. They include settings for query execution, memory management, compaction, caching, and more. Amazon Timestream provides default parameter groups, and you can create custom ones.

How do I create a custom parameter group?

Use the AWS CLI or the AWS Management Console to create a parameter group. Specify the edition (Core or Enterprise) and the parameters you want to customize. For example:

```
aws timestream-influxdb create-db-parameter-group \
  --name "my-custom-pg" \
  --description "Custom parameter group" \
  --parameters '{
    "InfluxDBv3Enterprise": {
      "queryFileLimit": 500,
      "queryLogSize": 2000
    }
  }'
```

Can I modify a parameter group after creation?

No. Parameter groups are immutable once created. To change parameters, create a new parameter group and assign it to your cluster using the `update-db-cluster` command. The update applies immediately and reboots your instance.

Are parameter changes applied at runtime?

No. All parameters are startup-only. There is no runtime reconfiguration. To apply changes, the cluster must be restarted with the updated parameter group.

Which parameters should I avoid changing?

Some parameters must not be changed after initial cluster setup because they affect how data is physically organized on disk. Changing them can cause data corruption or query failures. These include:

- `gen1-duration` – Controls first-level compaction window (default: 10 minutes)
- `compaction-gen2-duration` – Controls second-level compaction window
- `compaction-multipliers` – Controls higher-level compaction ratios

When creating a new parameter group, always keep these values identical to your current configuration. For the full list of parameters and their impact, see the section called "Detailed Parameter Reference".

What is the best practice for creating a parameter group?

Start from a default parameter group or clone an existing one, then modify only the specific parameters you need to change. Cloning is available through the AWS Management Console and ensures you inherit all safe defaults. Only adjust parameters you understand and have tested, and always preserve the compaction-related parameters (`gen1-duration`, `compaction-gen2-duration`, `compaction-multipliers`) unchanged.

How should I size parameter values for my instance type?

Parameter values should be tuned based on your instance type's vCPU and memory capacity. Key guidelines:

- **DataFusion threads** – Set to the number of vCPUs on your instance (for example, 16 for `db.influx.4xlarge`).
- **DataFusion max parquet fanout** – Scale with instance size: 250–500 for small instances, up to 5,000–10,000 for 12xlarge and above.

- **exec-mem-pool-bytes** – Keep at the default of 20% for mixed workload nodes. For query-only nodes, you can increase up to 70%.
- **parquet-mem-cache-size** – Keep at the default of 20% for most workloads. Increase to 25% for db.influx.4xlarge and above.

For detailed per-instance sizing tables, see the section called "Detailed Parameter Reference".

Schema design FAQ for Amazon Timestream for InfluxDB 3

Questions about designing schemas for Amazon Timestream for InfluxDB 3, including tags, fields, and best practices for performance. For detailed guidance, see the section called "InfluxDB schema design recommendations".

What is the difference between tags and fields?

Tags are indexed metadata used for grouping and filtering. Fields contain the actual measured values. Tags are part of the primary key and are stored as strings. Fields support multiple data types (float, integer, string, boolean) and are not indexed. Use tags for values you frequently filter or group by, and fields for measured data.

How does InfluxDB 3 handle high-cardinality data?

InfluxDB 3's columnar Parquet storage and Apache Arrow processing are specifically designed for high-cardinality workloads. Unlike previous versions, InfluxDB 3 does not suffer from performance degradation with high-cardinality tag values because it uses a fundamentally different storage architecture.

What are the schema design best practices?

Key recommendations include: keep tag cardinality manageable, use meaningful measurement names, avoid encoding data in measurement names, use consistent tag naming conventions, and design your schema around your most common query patterns. See the section called "InfluxDB schema design recommendations" for detailed guidance, and the InfluxDB 3 schema design best practices for additional recommendations.

Processing engine FAQ for Amazon Timestream for InfluxDB 3

Questions about extending Amazon Timestream for InfluxDB 3 with the built-in processing engine and Python plugins. For the complete guide, see the section called "Processing engine plugins".

What is the processing engine?

The processing engine lets you run custom Python plugins directly inside your InfluxDB 3 cluster. Plugins can respond to data writes, run on a schedule, or be triggered by HTTP requests. This enables real-time data transformation, alerting, and automation without external infrastructure. For getting started with plugins, see the InfluxDB 3 Enterprise processing engine documentation.

What types of plugins are supported?

InfluxDB 3 supports three plugin trigger types: write triggers (execute when data is written), schedule triggers (execute on a cron-like schedule), and request triggers (execute in response to HTTP requests). InfluxData also provides certified plugins for common use cases such as downsampling, anomaly detection, and forecasting. For the full plugin catalog, see the InfluxDB 3 plugin library.

Can I run my own custom plugins?

Yes. In addition to the InfluxData certified plugins, you can run your own Python plugins hosted in a plugin repository that you control—public or private. You configure the repository on a DB parameter group, apply it to your cluster, and reference plugins in triggers with the `gh:` prefix. Custom plugins run on both Core and Enterprise editions. For details, see the section called "Custom plugins".

Custom plugins FAQ for Amazon Timestream for InfluxDB 3

Questions about running your own Python plugins from public and private repositories on the Amazon Timestream for InfluxDB 3 processing engine. For the complete guide, see the section called "Custom plugins".

What is a custom plugin?

A custom plugin is a Python plugin that you write (rather than one of the InfluxData certified plugins) and run on the Timestream for InfluxDB 3 processing engine. You attach it to a trigger to transform data, send alerts, aggregate data, or expose a custom HTTP endpoint.

Which editions support custom plugins?

Custom plugins run on both Timestream for InfluxDB 3 Core and Enterprise editions. Set the plugin repository parameters under `InfluxDBv3Core` or `InfluxDBv3Enterprise` in your parameter group to match your cluster's edition.

Do I need to enable custom plugins?

Custom plugins must be enabled for your AWS account. You then enable them per cluster by creating a DB parameter group that sets `pluginRepositoryUrl` (and, for a private repository, `pluginRepositorySecretArn`) and applying it to the cluster with `create-db-cluster` or `update-db-cluster`.

What is the difference between a public and a private repository?

Both are configured the same way – a `pluginRepositoryUrl` on the parameter group. For a private repository, you additionally store a read-only access token in a Secrets Manager secret named `InfluxDB-RepoToken-*` and reference its ARN as `pluginRepositorySecretArn`, so the engine can authenticate when it fetches your plugins.

How do I change the plugin repository for my cluster?

The repository parameters are immutable on an existing parameter group. Create a new parameter group with the new `pluginRepositoryUrl` (and secret ARN, if private), then apply it to the cluster with `update-db-cluster --db-parameter-group-identifier`. Applying a parameter group runs a cluster maintenance action.

Can I use plugins from more than one repository on the same cluster – for example, the public InfluxData repository and my own repository?

Yes. Set the `pluginRepositoryUrl` base to the raw-content host only (no owner or repository), and give each trigger the full owner/repository/branch/path in its `gh:` path. Note the security tradeoff: a host-only base lets the cluster load plugin code from any repository the host can serve, so use it only when you genuinely need multiple repositories. For the full configuration, examples, and security guidance, see the section called “Use multiple plugin repositories”.

In which AWS Regions is this available?

Custom plugins are available in all AWS Regions where Timestream for InfluxDB 3 is available unless noted otherwise.

Is there an additional charge for running custom plugins?

There is no separate charge for the custom plugin feature. Plugin execution consumes the compute and memory of the cluster instances it runs on, so heavier plugins can affect the resources available for ingestion and queries.

How do I add Python package dependencies my plugin needs?

You can use the Python standard library and the pre-installed, Amazon-vetted libraries (such as `numpy`, `pandas`, and `httpx`). The package manager is disabled, so you cannot install additional third-party packages. If a plugin imports a package that is not available, the trigger fails and the import error appears in `system.processing_engine_logs`.

How are my private repository credentials protected?

You store a repository access token as a plain string in a Secrets Manager secret named `InfluxDB-RepoToken-*`, and Amazon Timestream for InfluxDB reads that secret with its service-linked role to fetch your plugins. Use a least-privileged, read-only token and rotate it by updating the secret value. To restrict access to only the service, attach the resource policies in the section called "Restrict the secret and CMK to Timestream for InfluxDB (resource policies)". Never hard-code tokens or other secrets in plugin source.

What happens if my repository token expires or the repository becomes unreachable?

If the token expires, the secret is deleted, or the repository cannot be reached, plugin retrieval fails the next time the engine loads plugins. Keep the token valid and update the secret when you rotate it. Check `system.processing_engine_logs` for retrieval errors.

Can I use custom plugins on a multi-node cluster?

Yes. As with certified plugins, place plugins according to node roles: data write plugins on ingester nodes, HTTP request plugins on querier nodes, and scheduled plugins on any node with a scheduler. Keep plugin configuration identical across the relevant nodes. For details, see Distributed deployment considerations.

My custom plugin is not running. How do I troubleshoot it?

Query `system.processing_engine_logs` for the trigger name to see load and runtime errors, and confirm the trigger is enabled with `system.processing_engine_triggers`. Common causes are an incorrect `gh:` path, a missing Python dependency, or (for private repositories) a token that cannot read the repository. See the section called "Troubleshoot custom plugins".

Storage and billing FAQ for Amazon Timestream for InfluxDB 3

Questions about how Amazon Timestream for InfluxDB 3 stores data, manages backups, and handles billing. For pricing details, see the section called "DB Cluster Billing for Amazon Timestream for InfluxDB 3".

How does storage work in InfluxDB 3?

InfluxDB 3 decouples compute from storage. Data is persisted in Apache Parquet format on Amazon S3 object storage, providing virtually unlimited and cost-effective storage. The compute nodes handle query processing and data ingestion while Amazon S3 handles durable storage.

How is Amazon Timestream for InfluxDB 3 billed?

Billing is based on compute node hours (per instance type), Amazon S3 object storage volume (GB/month), and data transfer. I/O operations are bundled into compute pricing for the `db.influxIOIncluded` instance classes, providing predictable costs regardless of I/O patterns.

Does Amazon Timestream for InfluxDB 3 support automatic backups?

Yes. Amazon Timestream for InfluxDB 3 automatically backs up your database and keeps your database software up to date with the latest version. Because data is stored on Amazon S3, it benefits from built-in durability and redundancy.

What are customer-managed backups?

Customer-managed backups let you create, manage, and restore database backups on demand or on a schedule—without opening an AWS support ticket. The first backup takes a complete copy of your database, and subsequent backups are incremental (only changed data). You can also enable continuous backups for point-in-time restore. For details, see the section called “Backup and restore”.

What is the difference between service-managed backups and customer-managed backups?

Service-managed backups are automatic, managed by AWS, and require a support ticket to restore. Customer-managed backups give you self-service control—you choose when to back up, how long to keep them, and can restore directly from the console or CLI.

Can I restore a backup to the same cluster?

Yes. You can restore using `REPLACE_EXISTING` mode, which replaces the data on your existing cluster with data from the backup. This is a destructive operation—existing data is deleted and the cluster is unavailable during the restore. You can also restore to a new cluster using `NEW_RESOURCE` mode (recommended).

Does a backup include my most recently written data?

Data written within the last 15 minutes might not be included in the backup. To recover the most recent data, use point-in-time restore with a timestamp at least 15 minutes in the past. Point-in-time restore requires a `CONTINUOUS` backup configuration.

Are there additional charges for customer-managed backups?

No feature charge. You pay only for the Amazon S3 storage consumed by your backups. Incremental backups reduce costs by storing only changed data.

Security and access FAQ for Amazon Timestream for InfluxDB 3

Questions about securing your Amazon Timestream for InfluxDB 3 clusters and managing access control. For the complete security guide, see the section called "Overview".

Does InfluxDB 3 run inside my VPC?

Yes. Amazon Timestream for InfluxDB 3 clusters are deployed within your VPC. You specify VPC subnet IDs and security group IDs when creating a cluster, giving you full control over network access.

How do I control access to Amazon Timestream for InfluxDB 3 resources?

Use AWS Identity and Access Management (IAM) to control access to Amazon Timestream API operations, including creating, modifying, and deleting resources such as DB clusters, security groups, and parameter groups. AWS provides managed policies for common access patterns.

Can I use my own encryption key (CMK) for data at rest?

Yes. You can specify a customer managed AWS KMS key when creating an InfluxDB 3 cluster. Your key encrypts the Amazon S3 objects that store your database data. If you don't specify a key, data is encrypted with an AWS owned key by default. For details, see the section called "Customer managed keys".

Can I change the encryption key after cluster creation?

No. The encryption key can only be specified during cluster creation. To migrate to a different key, create a new cluster with the desired key and migrate your data.

What happens if I disable or delete my customer managed key?

The cluster becomes unavailable and read/write operations fail. If you re-enable the key, the cluster automatically recovers. If you permanently delete the key, the data is unrecoverable. Service-managed backups for InfluxDB 3 use a separate key and are not affected.