
Amazon Timestream

Developer Guide



Amazon Timestream: Developer Guide

Copyright © 2022 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

What is Amazon Timestream?	1
Timestream key benefits	1
Timestream use cases	2
Getting started with Timestream	2
How it works	3
Concepts	3
A summary of Timestream concepts	4
Architecture	4
Write architecture	5
Storage architecture	6
Query architecture	6
Cellular architecture	6
Writes	7
No upfront schema definition	8
Writing data (inserts and upserts)	8
Batching writes	15
Eventual consistency for reads	15
Storage	15
Queries	16
Data model	16
Scheduled queries	18
Accessing Timestream	19
Signing up for AWS	19
Create an IAM user with Timestream access	19
Getting an AWS access key (not required for console access)	21
Configuring your credentials (not required for console access)	21
Using the console	22
Tutorial	22
Create a database	23
Create a table	23
Run a query	23
Create a scheduled query	24
Delete a scheduled query	24
Delete a table	25
Delete a database	25
Edit a table	25
Edit a database	25
Using the AWS CLI	26
Downloading and configuring the AWS CLI	26
Using the AWS CLI with Timestream	26
Using the API	28
The endpoint discovery pattern	28
How it works	29
Implementing the endpoint discovery pattern	29
Using the AWS SDKs	30
Java	31
Java v2	32
Go	32
Python	33
Node.js	33
.NET	33
Getting started	34
Tutorial	34
Using the console	34

Using the SDKs	34
Sample application	35
Code samples	37
Write SDK client	37
Query SDK client	39
Create database	40
Describe database	43
Update a database	45
Delete database	48
List databases	51
Create table	54
Memory store writes	54
Magnetic store writes	57
Describe table	60
Update table	63
Delete table	65
List tables	68
Write data (inserts and upserts)	71
Writing batches of records	71
Writing batches of records with common attributes	77
Upserting records	82
Handling write failures	95
Run query	97
Paginating results	97
Parsing result sets	100
Accessing the query status	109
Cancel query	114
Create scheduled query	116
List scheduled query	126
Describe scheduled query	129
Execute scheduled query	131
Update scheduled query	133
Delete scheduled query	135
Using scheduled queries	138
Benefits	138
Use cases	139
Example	139
Concepts	140
Schedule expressions	142
Data model mappings	144
Example: Target measure name for multi-measure records	145
Example: Using measure name from scheduled query in multi-measure records	147
Example: Mapping results to different multi-measure records with different attributes	149
Example: Mapping results to single-measure records with measure name from query results	152
Example: Mapping results to single-measure records with query result columns as measure names	155
Notification messages	157
Error reports	160
Patterns and examples	162
Sample schema	163
Patterns	181
Examples	200
Tagging resources	215
Tagging restrictions	215
Tagging operations	215
Adding tags to new or existing databases and tables using the console	216
Security	217

Data protection	217
Encryption at rest	218
Encryption in transit	218
Key management	219
Internetwork traffic privacy	219
Identity and access management	219
Audience	220
Authenticating with identities	220
Managing access using policies	222
How Amazon Timestream works with IAM	223
AWS managed policies	227
Identity-based policy examples	235
Troubleshooting	247
Logging and monitoring	249
Monitoring tools	250
Monitoring with Amazon CloudWatch	251
Logging Timestream API calls with AWS CloudTrail	257
Resilience	258
Infrastructure security	259
Configuration and vulnerability analysis	259
Incident response	259
VPC endpoints	259
How VPC endpoints work with Timestream	260
Creating an interface VPC endpoint for Timestream	260
Creating a VPC endpoint policy for Timestream	262
Security best practices	262
Preventative security best practices	262
Working with other services	264
AWS Lambda	264
Build AWS Lambda functions using Amazon Timestream with Python	264
Build AWS Lambda functions using Amazon Timestream with JavaScript	265
Build AWS Lambda functions using Amazon Timestream with Go	265
Build AWS Lambda functions using Amazon Timestream with C#	265
AWS IoT Core	265
Prerequisites	266
Using the console	267
Using the CLI	267
Sample application	268
Video tutorial	268
Amazon Kinesis Data Analytics for Apache flink	268
Sample application	269
Video tutorial	269
Amazon Kinesis	269
Amazon MSK	269
Amazon QuickSight	270
Accessing Amazon Timestream from QuickSight	270
To create a new QuickSight data source connection for Timestream	270
Edit permissions for the QuickSight data source connection for Timestream	271
To create a new QuickSight dataset for Timestream	271
Create a new analysis for Timestream	272
Video tutorial	272
Amazon SageMaker	272
Grafana	273
Sample application	274
Video tutorial	274
Open source Telegraf	275
Installing Telegraf with the Timestream output plugin	275

Running Telegraf with the Timestream output plugin	275
Mapping Telegraf/InfluxDB metrics to Timestream	276
JDBC	278
Configuring the JDBC driver	278
Connection properties	279
JDBC URL examples	283
SSO with Okta	284
SSO with Azure AD	285
VPC endpoints	288
Best practices	289
Data modeling	289
Single table vs. multiple tables	290
Multi-measure records vs. single-measure records	290
Dimensions and measures	292
Using measure name with multi-measure records	294
Recommendations for partitioning multi-measure records	296
Security	299
Configuring Timestream	299
Data ingestion	300
Queries	301
Scheduled queries	302
Client applications and supported integrations	302
General	302
Metering and cost optimization	303
Writes	303
Calculating the write size of a time series event	303
Calculating the number of writes	304
Storage	305
Queries	305
Cost optimization	306
Troubleshooting	307
Handling WriteRecords throttles	307
Handling rejected records	307
Timestream specific error codes	307
Timestream write API errors	307
Timestream query API errors	308
Quotas	309
Default quotas	309
Supported data types	311
Naming constraints	311
Reserved keywords	312
System identifiers	314
Query language reference	315
Supported data types	315
Built-in time series functionality	317
Timeseries views	318
Time series functions	320
SQL support	326
SELECT	326
Subquery support	327
SHOW statements	327
DESCRIBE statements	328
Logical operators	328
Comparison operators	329
Comparison functions	330
greatest()	330
least()	330

ALL(), ANY() and SOME()	330
Conditional expressions	331
CASE statement	332
IF statement	332
COALESCE statement	333
NULLIF statement	333
TRY statement	333
Conversion functions	333
cast()	333
try_cast()	333
Mathematical operators	333
Mathematical functions	334
String operators	336
String functions	336
Array operators	338
Array functions	339
Bitwise functions	340
Regular expression functions	341
Date / time operators	343
Operations	343
Addition	344
Subtraction	344
Date / time functions	345
General and conversion date / time functions	345
Interval and duration functions	349
Formatting and parsing functions	351
Extraction functions	352
Aggregate functions	354
Window functions	358
Sample queries	360
Simple queries	360
Queries with time series functions	361
Queries with aggregate functions	365
API reference	368
Actions	368
Amazon Timestream Write	369
Amazon Timestream Query	409
Data Types	447
Amazon Timestream Write	448
Amazon Timestream Query	467
Common Errors	503
Common Parameters	505
Document history	508

What is Amazon Timestream?

Amazon Timestream 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 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's purpose-built query engine lets you access and analyze recent and historical data together, without having to specify its location. Amazon Timestream has built-in time series analytics functions, helping you identify trends and patterns in your data in near real-time. Timestream 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 also integrates with commonly used services for data collection, visualization, and machine learning. You can send data to Amazon Timestream using AWS IoT Core, Amazon Kinesis, Amazon MSK, and open source Telegraf. You can visualize data using Amazon QuickSight, Grafana, and business intelligence tools through JDBC. You can also use Amazon SageMaker with Timestream for machine learning.

Topics

- [Timestream key benefits \(p. 1\)](#)
- [Timestream use cases \(p. 2\)](#)
- [Getting started with Timestream \(p. 2\)](#)

Timestream key benefits

The key benefits of Amazon Timestream are:

- *Serverless with auto-scaling* - With Amazon Timestream, there are no servers to manage and no capacity to provision. As the needs of your application change, Timestream automatically scales to adjust capacity.
- *Data lifecycle management* - Amazon Timestream 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, you no longer need to use disparate tools to access recent and historical data. Amazon Timestream'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 also supports advanced aggregates, window functions, and complex data types such as arrays and rows.
- *Always encrypted* - Amazon Timestream ensures that your time series data is always encrypted, whether at rest or in transit. Amazon Timestream 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 use cases

Examples of a growing list of use cases for Timestream 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

We recommend that you begin by reading the following sections:

- [Tutorial \(p. 34\)](#) - To create a database populated with sample data sets and run sample queries.
- [Amazon Timestream concepts \(p. 3\)](#) - To learn essential Timestream concepts.
- [Accessing Timestream \(p. 19\)](#) - To learn how to access Timestream using the console, AWS CLI, or API.
- [Quotas \(p. 309\)](#) - To learn about quotas on the number of Timestream components that you can provision.

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

- [Using the AWS SDKs \(p. 30\)](#)
- [Query language reference \(p. 315\)](#)

How it works

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

After you read this introduction, see the [Accessing Timestream \(p. 19\)](#) sections, to learn how to access Timestream using the Console, CLI or SDKs.

Topics

- [Amazon Timestream concepts \(p. 3\)](#)
- [Architecture \(p. 4\)](#)
- [Writes \(p. 7\)](#)
- [Storage \(p. 15\)](#)
- [Queries \(p. 16\)](#)
- [Scheduled queries \(p. 18\)](#)

Amazon Timestream 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.

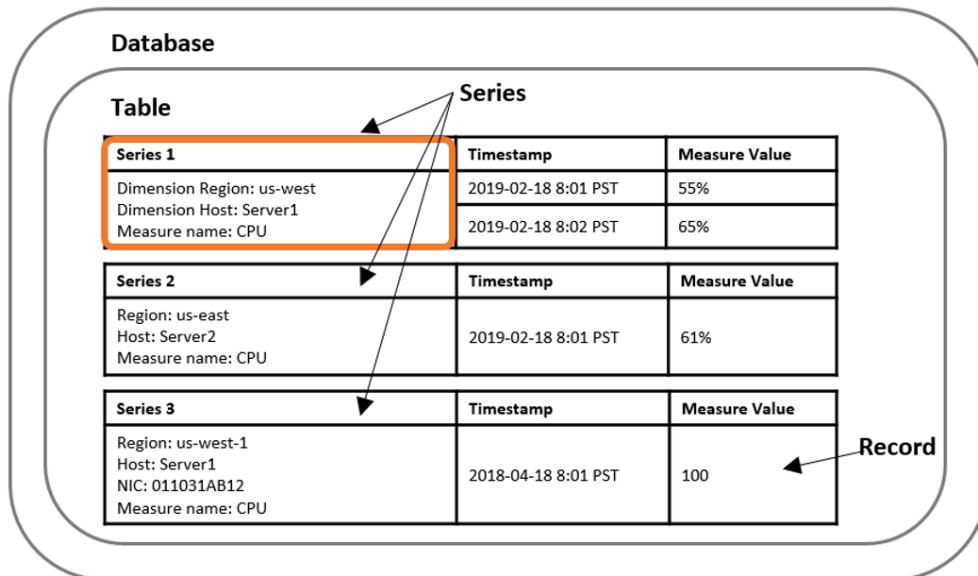
Following are the key concepts of Timestream.

- **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.
- **Timestamp** - *Indicates when a measure was collected for a given record.* Timestream 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 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**.

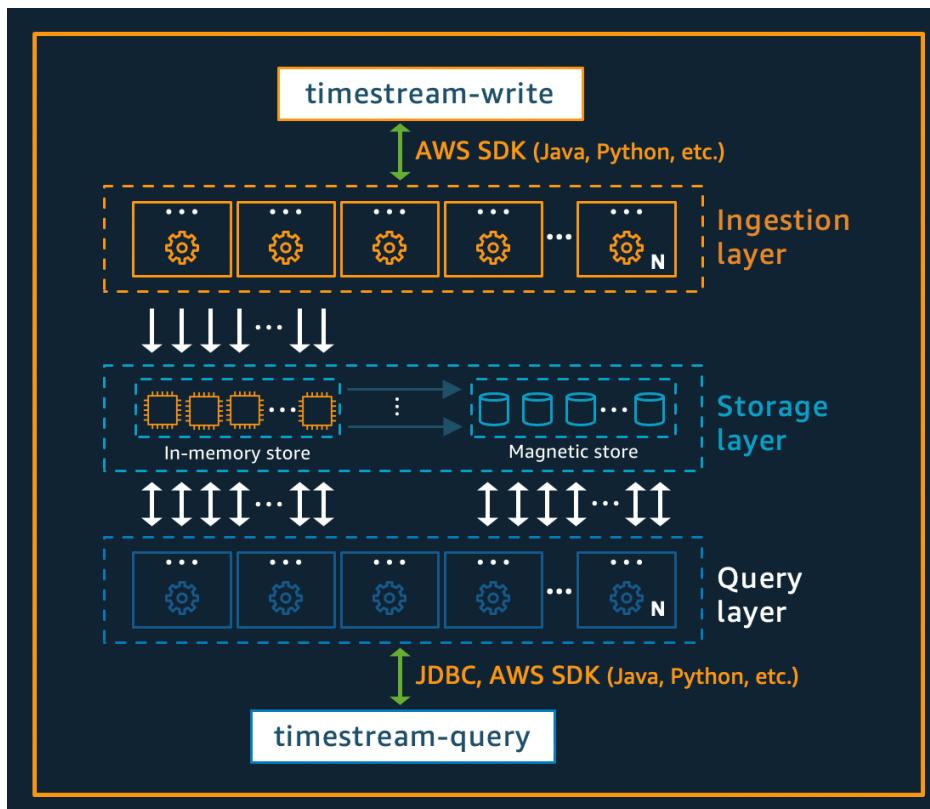


Architecture

Amazon Timestream 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 sub-system, 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. You can read more about each topic below.

Topics

- [Write architecture \(p. 5\)](#)
- [Storage architecture \(p. 6\)](#)
- [Query architecture \(p. 6\)](#)
- [Cellular architecture \(p. 6\)](#)



Write architecture

When writing time-series data Amazon Timestream 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 supports writing data directly into the magnetic store, for applications generating lower throughput late arrival data. Late arrival 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 automatically indexes and partitions data before writing it to storage. A single Timestream 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 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 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", since they represent divisions of a two-dimensional space which are designed to be of similar size. Timestream 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 is designed to automatically manage the lifecycle of time series data. Timestream offers two data stores—an in-memory store and a cost-effective magnetic store, and it 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 allows setting a data retention policy on the memory store for this purpose. Data writes for late arrival 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, other than some configuration, the data lifecycle management occurs seamlessly behind the scenes.

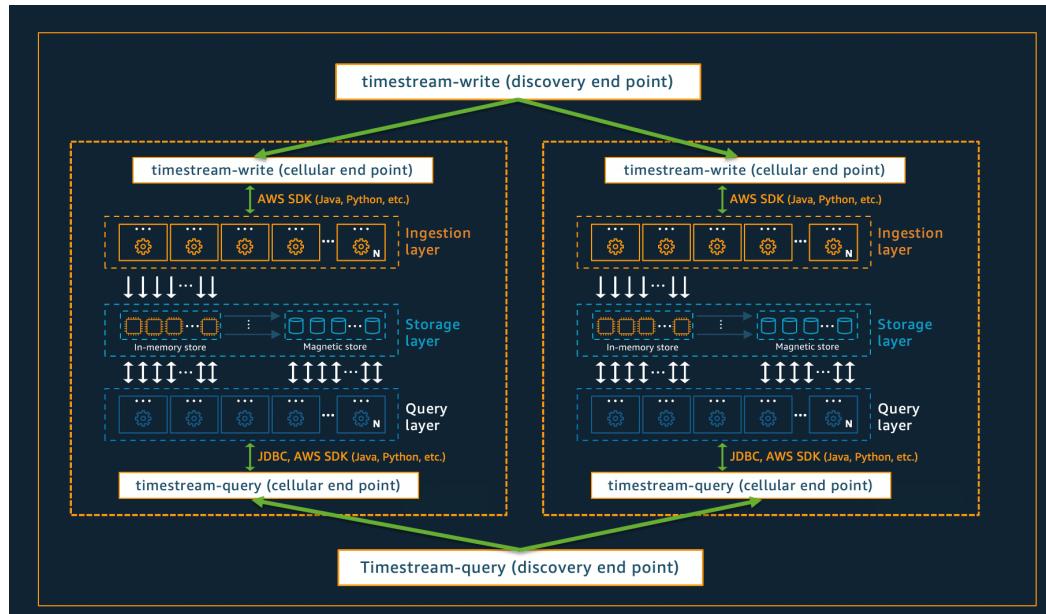
Query architecture

Timestream 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 data sets is achieved through massive parallelism, both on the query execution fleet and the storage fleets of the system. The ability to analyze massive amounts of data quickly and efficiently is one of Timestream's greatest strengths. A single query executing over terabytes or even petabytes of data may have thousands of machines working on it all at the same time.

Cellular architecture

To ensure that Timestream 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 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 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 or directly accessing REST APIs for Timestream, you will need to interact directly with the cellular endpoints. For guidance on how to do so, see [VPC Endpoints \(p. 259\)](#) for instructions on how to set up VPC Endpoints, and [Endpoint Discovery Pattern \(p. 28\)](#) for instructions on direct invocation of the REST APIs.

Writes

You can collect time series data from connected devices, IT systems, and industrial equipment, and write it into Timestream. Timestream 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 offers you with a flexible schema that auto detects the column names and data types for your Timestream 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.

Timestream 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.
VARCHAR	Variable length character data with an optional maximum length. The maximum limit is 2KB.

Data type	Description
MULTI	Data type for multi-measure records. This data type includes one or more measures of type BIGINT, BOOLEAN, DOUBLE, VARCHAR, and TIMESTAMP.
TIMESTAMP	<p>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).</p> <p><i>YYYY-MM-DD hh:mm:ss.ssssssss</i></p> <p>Writes support timestamps in the range 1970-01-01 00:00:00.000000000 to 2262-04-11 23:47:16.854775807.</p>

Note

Timestream supports eventual consistency semantics for reads. This means that when you query data immediately after writing a batch of data into Timestream, 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 \(p. 30\)](#), [AWS CLI \(p. 26\)](#), or through [AWS Lambda \(p. 264\)](#), [AWS IoT Core \(p. 265\)](#), [Amazon Kinesis Data Analytics for Apache flink \(p. 268\)](#), [Amazon Kinesis \(p. 269\)](#), [Amazon MSK \(p. 269\)](#) and [Open source Telegraf \(p. 275\)](#).

Topics

- [No upfront schema definition \(p. 8\)](#)
- [Writing data \(inserts and upserts\) \(p. 8\)](#)
- [Batching writes \(p. 15\)](#)
- [Eventual consistency for reads \(p. 15\)](#)

No upfront schema definition

Before sending data into Amazon Timestream, you must create a database and a table using the AWS Management Console, Timestream's SDKs, or the Timestream APIs. For more information, see [Create a database \(p. 23\)](#) and [Create a table \(p. 23\)](#). While creating the table, you do not need to define the schema up front. Amazon Timestream 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 enables you to insert and *upsert* data. By default, writes in Amazon Timestream 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 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. When the record is updated, it is updated in an idempotent manner.

Writing data into the memory store and the magnetic store

Amazon Timestream 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 arrival data is data with a timestamp earlier than the current time. You must explicitly enable the ability to write late arrival data into the magnetic store by enabling magnetic store writes for the table.

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

Amazon Timestream 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 using this format, Timestream 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 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 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 to identify more repeatable data in a single write request (where applicable), and charge only once for repeated data.

Topics

- [Multi-measure records \(p. 9\)](#)
- [Writing data with a timestamp that exists in the past or in the future \(p. 14\)](#)

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 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 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 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, 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	measure_value::double
2021-09-07 21:48:44.000000000	12345678	iPhone 11	14.8	us-east-1	video_startup_time	200ms	
2021-09-07 21:48:44.000000000	12345678	iPhone 11	14.8	us-east-1	rebuffering_ratio	0.5	
2021-09-07 21:48:44.000000000	12345678	iPhone 11	14.8	us-east-1	video_playback_failures	10	
2021-09-07 21:48:44.000000000	12345678	iPhone 11	14.8	us-east-1	average_frame_rate	0.85	
2021-09-07 21:53:44.000000000	12345678	iPhone 11	14.8	us-east-1	video_startup_time	500ms	
2021-09-07 21:53:44.000000000	12345678	iPhone 11	14.8	us-east-1	rebuffering_ratio	1.5	
2021-09-07 21:53:44.000000000	12345678	iPhone 11	14.8	us-east-1	video_playback_failures	10	
2021-09-07 21:53:44.000000000	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_rebuffering_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, 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")
                .withValueType(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))
        .withName("video_metrics")
        .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_id	device_ty	os_versio	region	measure_	video_sta	rebufferi	video_	average_frame_rate
2021-09-07 21:48:44	2345678	iPhone	14.8	us-east-1	video_me	100	0.5	0	0.85
2021-09-07 21:53:44	2345678	iPhone	14.8	us-east-1	video_me	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 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 arrival 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 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 datatype 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 does not index on the values of type TIMESTAMP in a multi-measure record.

Note

The `TIMESTAMP` datatype is supported only for multi-measure records.

Batching writes

Amazon Timestream 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 \(p. 303\)](#) in the Metering and Pricing section for more details.

Note

Your 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.

Eventual consistency for reads

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

Storage

Timestream 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 arrival data writes, long term data storage, and fast analytical queries.

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 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 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. The retention values of the memory store and the magnetic store cumulatively define the amount of time your data will be stored in Timestream. This means that for the above scenario, from the time of data arrival, the data is stored in Timestream 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 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 is always encrypted by default. This applies to data in transit and at rest. Furthermore, Timestream 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, 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. Timestream's purpose-built, adaptive query engine 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 to retrieve time series data from one or more tables. You can access the metadata information for databases and tables. Timestream's SQL also supports built-in functions for time series analytics. You can refer to the [Query language reference \(p. 315\)](#) reference for additional details.

Timestream is designed to have a fully decoupled data ingestion, storage, and query architecture where each component can scale independent of other components, allowing it to offer virtually infinite scale for an application's needs. This means that Timestream 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, Timestream's query latency remains mostly unchanged. This is because Timestream's 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 \(p. 16\)](#)
- [Time series model \(p. 17\)](#)

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.

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_na	measure_va	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	cpu utilization	35.0	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	cpu utilization	38.2	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	cpu utilization	45.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	memory utilization	54.9	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	memory utilization	42.6	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	memory utilization	33.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	network bytes	34,400	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	network bytes	1,500	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	network bytes	6,600	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_r	cpu_utilization	memory_usage	network_bytes
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	cpu utilization	35.0	54.9	34,400
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	cpu utilization	38.2	42.6	1,500
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d1234567890abcdef0	0	cpu utilization	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 \(p. 315\)](#) 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-1234567890abcdef	[{time: 2019-12-04 19:00:00.000000000, value: 35}, {time: 2019-12-04 19:00:01.000000000, value: 38.2}, {time: 2019-12-04 19:00:02.000000000, value: 45.3}]

Scheduled queries

The scheduled query feature in Amazon Timestream 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 \(p. 138\)](#).

Accessing Timestream

You can access Timestream using the console, CLI or the API. Before accessing Timestream, you need to do the following:

1. [Sign up for AWS. \(p. 19\)](#)
2. Create an IAM user with Timestream access. For more information, see [Create an IAM user with Timestream access \(p. 19\)](#).
3. [Get an AWS access key \(p. 21\)](#) (not required for Console access).
4. [Configure your credentials \(p. 21\)](#) (not required for Console access).

Signing up for AWS

To use the Timestream service, you must have an AWS account. If you don't already have an account, you are prompted to create one when you sign up.

To sign up for AWS

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

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](#).

Create an IAM user with Timestream access

When you signed up for AWS, you created an AWS account using an email address and password. Those are your AWS account root user credentials. As a best practice, you should not use your AWS account root user credentials to access AWS. Nor should you give your credentials to anyone else. Instead, create individual users for those who need access to your AWS account. Create an AWS Identity and Access Management (IAM) user for yourself also. Give that user administrative permissions, and use that IAM user for all your work. For information about how to do this, see [Creating Your First IAM Admin User and Group](#) in the *IAM User Guide*.

If you're an account owner or administrator and want to know more about IAM, see the product description at <https://aws.amazon.com/iam> or the technical documentation in the [IAM User Guide](#).

To create an administrator user, choose one of the following options.

Choose one way to manage your administration	To	By	You can also
In IAM Identity Center (Recommended)	Use short-term credentials to access AWS. This aligns with the security best practices. For information about best practices, see Security best practices in IAM in the <i>IAM User Guide</i> .	Following the instructions in Getting started in the <i>AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide</i> .	Configure programmatic access by Configuring the AWS CLI to use AWS IAM Identity Center (successor to AWS Single Sign-On) in the <i>AWS Command Line Interface User Guide</i> .
In IAM (Not recommended)	Use long-term credentials to access AWS.	Following the instructions in Creating your first IAM admin user and user group in the <i>IAM User Guide</i> .	Configure programmatic access by Managing access keys for IAM users in the <i>IAM User Guide</i> .

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

```
{
  "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": "*"
    }
  ]
}
```

Getting an AWS access key (not required for console access)

Before you can access Timestream programmatically, you must have an AWS access key. You don't need an access key if you plan to use the Timestream console only.

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them from the AWS Management Console. As a best practice, do not use the AWS account root user access keys for any task where it's not required. Instead, [create a new administrator IAM user](#) with access keys for yourself.

The only time that you can view or download the secret access key is when you create the keys. You cannot recover them later. However, you can create new access keys at any time. You must also have permissions to perform the required IAM actions. For more information, see [Permissions required to access IAM resources](#) in the *IAM User Guide*.

To create access keys for an IAM user

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane, choose **Users**.
3. Choose the name of the user whose access keys you want to create, and then choose the **Security credentials** tab.
4. In the **Access keys** section, choose **Create access key**.
5. To view the new access key pair, choose **Show**. You will not have access to the secret access key again after this dialog box closes. Your credentials will look something like this:
 - Access key ID: AKIAIOSFODNN7EXAMPLE
 - Secret access key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
6. To download the key pair, choose **Download .csv file**. Store the keys in a secure location. You will not have access to the secret access key again after this dialog box closes.

Keep the keys confidential in order to protect your AWS account and never email them. Do not share them outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.

7. After you download the .csv file, choose **Close**. When you create an access key, the key pair is active by default, and you can use the pair right away.

Related topics

- [What is IAM?](#) in the *IAM User Guide*
- [AWS security credentials](#) in *AWS General Reference*

Configuring your credentials (not required for console access)

Before you can access Timestream programmatically or through the AWS CLI, you must configure your credentials to enable authorization for your applications.

There are several ways to do this. For example, you can manually create the credentials file to store your AWS access key ID and secret access key. You can also use the `aws configure` command of the AWS CLI to automatically create the file. Alternatively, you can use environment variables.

To connect to Timestream with the AWS SDK for Java, you must provide AWS credentials. The Timestream Java client tries to find AWS credentials by using the *default credential provider chain* implemented by the `DefaultAWSCredentialsProviderChain` class. For more information about using the default credential provider chain, see [Working with AWS Credentials](#) in the AWS SDK for Java Developer Guide.

To install and configure the AWS CLI, see [Accessing Amazon Timestream using the AWS CLI \(p. 26\)](#).

You can access Timestream using the AWS Management Console, CLI, or the Timestream API.

- [Using the Console \(p. 22\)](#)
- [Using the CLI \(p. 26\)](#)
- [Using the API \(p. 28\)](#)

Using the console

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

Topics

- [Tutorial \(p. 22\)](#)
- [Create a database \(p. 23\)](#)
- [Create a table \(p. 23\)](#)
- [Run a query \(p. 23\)](#)
- [Create a scheduled query \(p. 24\)](#)
- [Delete a scheduled query \(p. 24\)](#)
- [Delete a table \(p. 25\)](#)
- [Delete a database \(p. 25\)](#)
- [Edit a table \(p. 25\)](#)
- [Edit a database \(p. 25\)](#)

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 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 \(p. 23\)](#).
 - **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 \(p. 23\)](#).
4. In the left pane, select the database created in [Create a table \(p. 23\)](#).
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
```

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. On the select destination table and query name page, enter the following.
 - **Database name**—Select the name of the database created in [Create a database \(p. 23\)](#).
 - **Table name**—Select the name of the table created in [Create a table \(p. 23\)](#).
 - **Query name**—Enter a query name.
5. Click on **Next**.
6. On the configure query statement and table schema page, enter the following:
 - Enter a valid **Query statement**, then choose **Validate**.
 - Set the **Destination table schema** for any underdefined attributes.
7. Click on **Next**.
8. On the configure query query settings page, enter the following.
 - A **Run schedule**. Refer to [Schedule Expressions for Scheduled Queries](#) for more details on schedule expressions.
 - The **IAM role** that Timestream 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.
 - The optional customer managed **KMS key** that will be used to encrypt the query statement.
 - The **SNS Topic** that will be used to for notification.
 - The **Error report logging S3 location** that will be used to report errors.
9. Click on **Next**.
10. Click on **Create**.

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 \(p. 24\)](#).
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 \(p. 23\)](#).
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 \(p. 23\)](#).
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 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 operations within utility scripts.

Before you can use the AWS CLI with Timestream, you must get an access key ID and secret access key. For more information, see [Getting an AWS access key \(not required for console access\) \(p. 21\)](#).

For a complete listing of all the commands available for the Timestream Query API in the AWS CLI, see the [AWS CLI Command Reference](#).

For a complete listing of all the commands available for the Timestream Write API in the AWS CLI, see the [AWS CLI Command Reference](#).

Topics

- [Downloading and configuring the AWS CLI \(p. 26\)](#)
- [Using the AWS CLI with Timestream \(p. 26\)](#)

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

The command line format consists of an Amazon Timestream 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 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\", \"ValueType\": \"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\": \"serrors\",\
\"EncryptionOption\": \"SSE_S3\"\
}}"
}

```

Using the API

In addition to the [SDKs \(p. 30\)](#), Amazon Timestream 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'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 REST API endpoint discovery pattern is necessary:

- You are using [VPC endpoints \(AWS PrivateLink\) with Timestream \(p. 259\)](#)
- 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 \(p. 29\)](#)
- [Implementing the endpoint discovery pattern \(p. 29\)](#)

How the endpoint discovery pattern works

Timestream is built using a [cellular architecture \(p. 6\)](#) 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 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 \(p. 29\)](#).

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 \(p. 30\)](#).

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 regional endpoints recognize.
- The response contains a list of endpoints to make Timestream 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 \(p. 37\)](#).

Topics

- [Java \(p. 31\)](#)
- [Java v2 \(p. 32\)](#)
- [Go \(p. 32\)](#)

- [Python \(p. 33\)](#)
- [Node.js \(p. 33\)](#)
- [.NET \(p. 33\)](#)

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 \(p. 37\)](#).

Prerequisites

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

1. Follow the AWS setup instructions in [Accessing Timestream \(p. 19\)](#).
 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 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 \(p. 37\)](#).

Prerequisites

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

1. Follow the AWS setup instructions in [Accessing Timestream \(p. 19\)](#).
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 \(p. 37\)](#).

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 \(p. 37\)](#).

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 \(p. 37\)](#).

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 \(p. 37\)](#).

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, 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 \(p. 34\)](#)
- [Sample application \(p. 35\)](#)

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:

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.
 - **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.

Using the SDKs

Timestream 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 sample applications following the instructions from [GitHub](#).
2. Configure the AWS SDK to connect to Amazon Timestream following the instructions described in [Using the AWS SDKs \(p. 30\)](#).
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 sample applications](#) following the instructions from [GitHub](#).
2. Configure the AWS SDK to connect to Timestream following the instructions described in [Getting Started with Java \(p. 31\)](#).
3. Run the [Java sample application](#) following the instructions described [here](#)

Java v2

1. Clone the GitHub repository [Timestream sample applications](#) following the instructions from [GitHub](#).
2. Configure the AWS SDK to connect to Amazon Timestream following the instructions described in [Getting Started with Java v2 \(p. 32\)](#).
3. Run the [Java 2.0 sample application](#) following the instructions described [here](#)

Go

1. Clone the GitHub repository [Timestream sample applications](#) following the instructions from [GitHub](#).
2. Configure the AWS SDK to connect to Amazon Timestream following the instructions described in [Getting Started with Go \(p. 32\)](#).
3. Run the [Go sample application](#) following the instructions described [here](#)

Python

1. Clone the GitHub repository [Timestream sample applications](#) following the instructions from [GitHub](#).
2. Configure the AWS SDK to connect to Amazon Timestream following the instructions described in [Python \(p. 33\)](#).
3. Run the [Python sample application](#) following the instructions described [here](#)

Node.js

1. Clone the GitHub repository [Timestream sample applications](#) following the instructions from [GitHub](#).
2. Configure the AWS SDK to connect to Amazon Timestream following the instructions described in [Getting Started with Node.js \(p. 33\)](#).
3. Run the [Node.js sample application](#) following the instructions described [here](#)

.NET

1. Clone the GitHub repository [Timestream sample applications](#) following the instructions from [GitHub](#).
2. Configure the AWS SDK to connect to Amazon Timestream following the instructions described in [Getting Started with .NET \(p. 33\)](#).
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 \(p. 37\)](#)
- [Query SDK client \(p. 39\)](#)
- [Create database \(p. 40\)](#)
- [Describe database \(p. 43\)](#)
- [Update database \(p. 45\)](#)
- [Delete database \(p. 48\)](#)
- [List databases \(p. 51\)](#)
- [Create table \(p. 54\)](#)
- [Describe table \(p. 60\)](#)
- [Update table \(p. 63\)](#)
- [Delete table \(p. 65\)](#)
- [List tables \(p. 68\)](#)
- [Write data \(inserts and upserts\) \(p. 71\)](#)
- [Run query \(p. 97\)](#)
- [Cancel query \(p. 114\)](#)
- [Create scheduled query \(p. 116\)](#)
- [List scheduled query \(p. 126\)](#)
- [Describe scheduled query \(p. 129\)](#)
- [Execute scheduled query \(p. 131\)](#)
- [Update scheduled query \(p. 133\)](#)
- [Delete scheduled query \(p. 135\)](#)

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 \(p. 35\)](#).

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 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 \(p. 35\)](#).

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 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 \(p. 35\)](#).

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 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 \(p. 35\)](#).

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.Arns}`);
} 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 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.Arns}`);
  },
  (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.Arns}");
    }
    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 \(p. 35\)](#).

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 {
        timestampWriteClient.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 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 \(p. 35\)](#).

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 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 \(p. 35\)](#).

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 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 \(p. 54\)](#)
- [Magnetic store writes \(p. 57\)](#)

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 \(p. 35\)](#).

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).build()

    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 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 \(p. 35\)](#).

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 {
        timestampWriteClient.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 \(p. 35\)](#).

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.Arns}`);
    } 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 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.Arns}`);
        },
        (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.Arns}");
    }
    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 \(p. 35\)](#).

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).build()  
  
    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 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 \(p. 35\)](#).

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 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 exist.`);
            } 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 \(p. 35\)](#).

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 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(...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 \(p. 71\)](#)
- [Writing batches of records with common attributes \(p. 77\)](#)
- [Upserting records \(p. 82\)](#)
- [Handling write failures \(p. 95\)](#)

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 \(p. 304\)](#) 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 \(p. 35\)](#).

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 =
        timestampWriteClient.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"])

```

Node.js

```

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. Using common attributes with batching can further optimize the cost of writes as described in [Calculating the number of writes \(p. 304\)](#).

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 \(p. 35\)](#).

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.getSdkHttpResponseMetadata().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"])

```

Node.js

```

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 \(p. 35\)](#).

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 is 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 =
        timestampWriteClient.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 is 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("First-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 is 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)

```

Node.js

```

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 is 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());
    }
}

```

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`.

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 \(p. 35\)](#).

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

```

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 \(p. 97\)](#)
- [Parsing result sets \(p. 100\)](#)
- [Accessing the query status \(p. 109\)](#)

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.

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 \(p. 35\)](#).

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

```
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 \(p. 35\)](#).

Java

```

private static final DateTimeFormatter TIMESTAMP_FORMATTER =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.SSSSSSSS");
private static final DateTimeFormatter DATE_FORMATTER =
DateTimeFormatter.ofPattern("yyyy-MM-dd");
private static final DateTimeFormatter TIME_FORMATTER =
DateTimeFormatter.ofPattern("HH:mm:ss.SSSSSSSS");

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.getTimeSeriesMeasureColumnInfo() != 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().getTimeSeriesMeasureColumnInfo(),
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.isNull() != null & datum.isNull()) {
            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
        }
    }
    return value
}

```

```

        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

```

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 \(p. 35\)](#).

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 =
    timestampQueryClient.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

```

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 \(p. 114\)](#).

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 \(p. 35\)](#).

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

```
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 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(SCHEDED_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();
}

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 = "scheduledquerysampleerrorbucket" + ''.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

```

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.Arns);
    return data.Arns;
} 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> sourceColToMeasureValueType =
        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

```
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

```

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

```
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

```
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

```
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 scheduled queries in Timestream

The scheduled query feature in Amazon Timestream 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 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 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 \(p. 138\)](#)
- [Scheduled query use cases \(p. 139\)](#)
- [Example: Using real-time analytics to detect fraudulent payments and make better business decisions \(p. 139\)](#)
- [Scheduled query concepts \(p. 140\)](#)
- [Schedule expressions for scheduled queries \(p. 142\)](#)
- [Data model mappings for scheduled queries \(p. 144\)](#)
- [Scheduled query notification messages \(p. 157\)](#)
- [Scheduled query error reports \(p. 160\)](#)
- [Scheduled query patterns and examples \(p. 162\)](#)

Scheduled query benefits

The following are the benefits of scheduled queries:

- **Operational ease** – Scheduled queries are serverless and fully managed. All you need to do is define the required inputs, and Amazon Timestream will take care of the rest.
- **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 tables and can be used with all of the services and tools that work with Timestream. 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 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 processes ~100K transactions per minute. Each transaction stored in Amazon Timestream 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 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 table. You can define a scheduled query using the full SQL surface area of Timestream, 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 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 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 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 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 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 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 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.

Schedule expressions for scheduled queries

You can create scheduled queries on an automated schedule by using Amazon Timestream 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	, - * /
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.
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.

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
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 supports flexible modeling of data in its tables and this same flexibility applies to results of scheduled queries that are materialized into another Timestream 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 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 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

Column	Type	Timestream attribute type
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 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 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"}]
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 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 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 attribute type
region	varchar	DIMENSION
measure_name	varchar	MEASURE_NAME
time	timestamp	TIMESTAMP
minCpuIdle	double	MULTI

Column	Type	Timestream attribute type
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"}]
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 table_name
GROUP BY region, hour, measure_name
ORDER BY hour, measure_name
  }"
```

```

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 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
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"}]

measure_name	data_type	Dimensions
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 `NONE`, 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",
      "TableArn" : "arn:aws:timestream:us-east-1:123456789012:table/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 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 when creating, deleting, running, or updating the state of a scheduled query.

Notification message name	Structure	Description
CreatingNotificationMessage	<pre>CreatingNotificationMessage { String arn; NotificationType type; }</pre>	<p>This notification message is sent before sending the response for <code>CreateScheduledQuery</code>. The scheduled query is enabled after sending this notification.</p> <p><i>arn</i> - The ARN of the scheduled query that is being created.</p>

Notification message name	Structure	Description
		<p><i>type</i> - SCHEDULED_QUERY_CREATING</p>
UpdateNotificationMessage	<pre>UpdateNotificationMessage { String arn; NotificationType type; QueryState state; }</pre>	<p>This notification message is sent when a scheduled query is updated. Timestream can disable the scheduled query, automatically, in case non-recoverable error is encountered, such as:</p> <ul style="list-style-type: none"> • 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 <p><i>arn</i> - The ARN of the scheduled query that is being updated.</p> <p><i>type</i> - SCHEDULED_QUERY_UPDATE</p> <p><i>state</i> - ENABLED or DISABLED</p>
DeleteNotificationMessage	<pre>DeletionNotificationMessage { String arn; NotificationType type; }</pre>	<p>This notification message is sent when a scheduled query has been deleted.</p> <p><i>arn</i> - The ARN of the scheduled query that is being created.</p> <p><i>type</i> - SCHEDULED_QUERY_DELETED</p>

Notification message name	Structure	Description
SuccessNotificationMessage	<pre>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 { S3ReportLocation s3ReportLocation; } S3ReportLocation { String bucketName; String objectKey; }</pre>	<p>This notification message is sent after the scheduled query is run and the results are successfully ingested.</p> <p><i>ARN</i> - The ARN of the scheduled query that is being deleted.</p> <p><i>NotificationType</i> - AUTO_TRIGGER_SUCCESS or MANUAL_TRIGGER_SUCCESS.</p> <p><i>nextInvocationEpochSecond</i> - The next time Timestream will run the scheduled query.</p> <p><i>runSummary</i> - Information about the scheduled query run.</p>

Notification message name	Structure	Description
FailureNotificationMessage	<pre>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; } S3ReportLocation { String bucketName; String objectKey; }</pre>	<p>This notification message is sent when a failure is encountered during a scheduled query run or when ingesting the query results.</p> <p><i>arn</i> - The ARN of the scheduled query that is being run.</p> <p><i>type</i> - AUTO_TRIGGER_FAILURE or MANUAL_TRIGGER_FAILURE.</p> <p><i>runSummary</i> - Information about the scheduled query run.</p>

Scheduled query error reports

This section describes the format of the error reports that are generated by Timestream, when an error is encountered during the running a scheduled query.

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
}
```

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
}
```

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

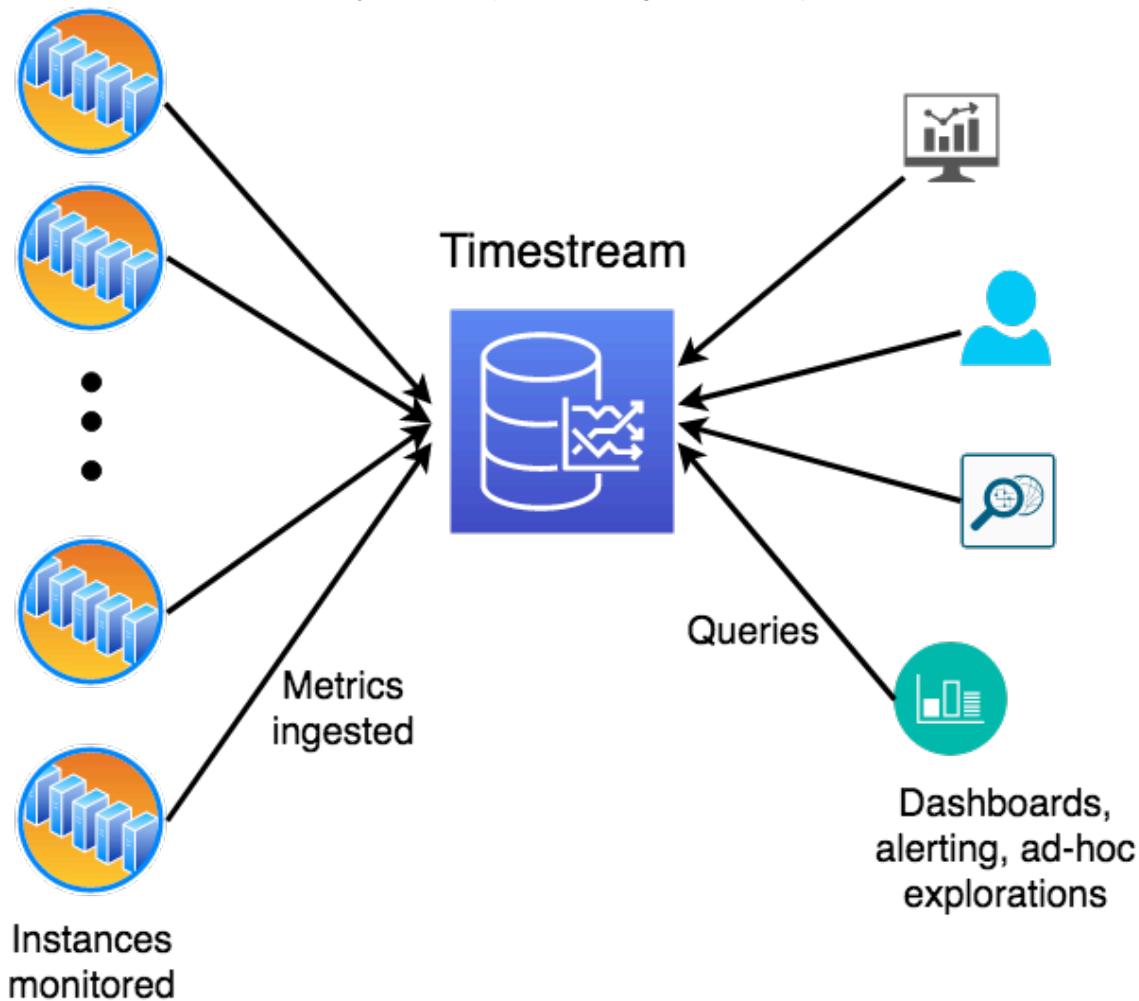
This section describes the usage patterns for scheduled queries as well as end-to-end examples.

Topics

- [Scheduled queries sample schema \(p. 163\)](#)
- [Scheduled query patterns \(p. 181\)](#)
- [Scheduled query examples \(p. 200\)](#)

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. 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. 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 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 \(p. 164\)](#)
- [Single-measure records \(p. 167\)](#)

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 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
memory_free	double	MULTI
cpu_steal	double	MULTI
cpu_iowait	double	MULTI

Column	Type	Timestream attribute type
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
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": "avai"}, {"data_type": "varchar", "dimension_name": "micro"}, {"data_type": "varchar", "dimension_name": "insta"}, {"data_type": "varchar", "dimension_name": "proc"}, {"data_type": "varchar", "dimension_name": "jdk"}, {"data_type": "varchar", "dimension_name": "cell"}]

measure_name	data_type	Dimensions
		{"data_type": "varchar", "dimension_name": "region"}, {"data_type": "varchar", "dimension_name": "silo"}
metrics	multi	[{"data_type": "varchar", "dimension_name": "available"}, {"data_type": "varchar", "dimension_name": "micro"}, {"data_type": "varchar", "dimension_name": "instance"}, {"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"}]

Example Data

Single-measure records

Timestream 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 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"}]

measure_name	data_type	Dimensions
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'}, {'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'}]]
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'}]]
cpu_steam	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_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'}]]
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'}]]
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'}]]
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'}]]
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'}]]
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'}]]
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'}]]
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'}]
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'}]]
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'}]]
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

availability_zone	microservice_name	instance_name	process_name	os_version	jdk_version	Cell	region	Silo	instance_type	measure_name	Time	measure_value	measure_dimensions	subject
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_high	2024-07-20T17:34:57	20.87169		
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_idle	2024-07-20T17:34:57	23.46266		
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_low	2024-07-20T17:34:57	20.10226		
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_nice	2024-07-20T17:34:57	20.63013		

available	micro	instance	process	os_version	jdk_version	Cell	region	Silo	instance_type	measurements	Time	measurements	measurements	measurements	measurements
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com	AL2012			eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_si	34:57.20.16441				
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com	AL2012			eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_st	34:57.20.10729				
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com	AL2012			eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_sy	34:57.20.45709				
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com	AL2012			eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	cpu_us	34:57.294.20448				
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com	AL2012			eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	disk_free	34:57.272.51895				
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com	AL2012			eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	disk_io	34:55.281.73383				
eu-west-1-1	hercules	zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com	AL2012			eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	disk_io	34:55.277.11665				

available	micro	instance	process	os_version	jdk_version	Cell	region	Silo	instance	measure	Time	measure	measure	measure	measure	measure
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	latency	2024-07-24T15:57:289.42235					
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	latency	2024-07-24T15:57:250.08254					
eu-west-1-1	hercules	server zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		JDK_8	eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		gc_pause	2024-07-24T15:57:260.28679						
eu-west-1-1	hercules	server zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		JDK_8	eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		gc_reclaim	2024-07-24T15:57:275.28839						
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	latency	2024-07-24T15:57:286.7605					
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	latency	2024-07-24T15:57:291.1223					
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2	r5.4xlarge	memory	2024-07-24T15:57:291.56481					

available	micro	instance	processor	os_version	jdk_version	Cell	region	Silo	instance_id	measurements	Time	measurements	measurements	measurements	subject
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com			JDK_8	eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		memory	3456	218.95768			
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		r5.4xlarge	memory	3456	297.20523		
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		r5.4xlarge	memory	3456	212.37723		
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		r5.4xlarge	network	145725_1	102065		
eu-west-1-1	hercules	is zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com		AL2012		eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		r5.4xlarge	network	145725_2	104124		
eu-west-1-1	hercules	server zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com			JDK_8	eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		task_count	3456	22	69		
eu-west-1-1	hercules	server zaZswmJk-hercules-eu-west-1-cell-9-silo-2-00000027.amazonaws.com			JDK_8	eu-west-1-cell-9	eu-west-1	eu-west-1-cell-9-silo-2		task_end_time	345725_2			SUCCESS_WITH_RB	

Scheduled query patterns

In this section you will find some common patterns of how you can use Amazon Timestream 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 allow you to express your queries using the full SQL surface area of Timestream. Your query can include one or more source tables, perform aggregations or any other query allowed by Timestream's SQL language, and then materialize the results of the query in another destination table in Timestream. 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 \(p. 181\)](#)
- [Simple fleet-level aggregates \(p. 182\)](#)
- [Last point from each device \(p. 188\)](#)
- [Unique dimension values \(p. 192\)](#)
- [Handling late-arriving data \(p. 195\)](#)
- [Back-filling historical pre-computations \(p. 200\)](#)

Scenario

The following examples use a DevOps monitoring scenario which is outlined in [Scheduled queries sample schema \(p. 163\)](#).

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.

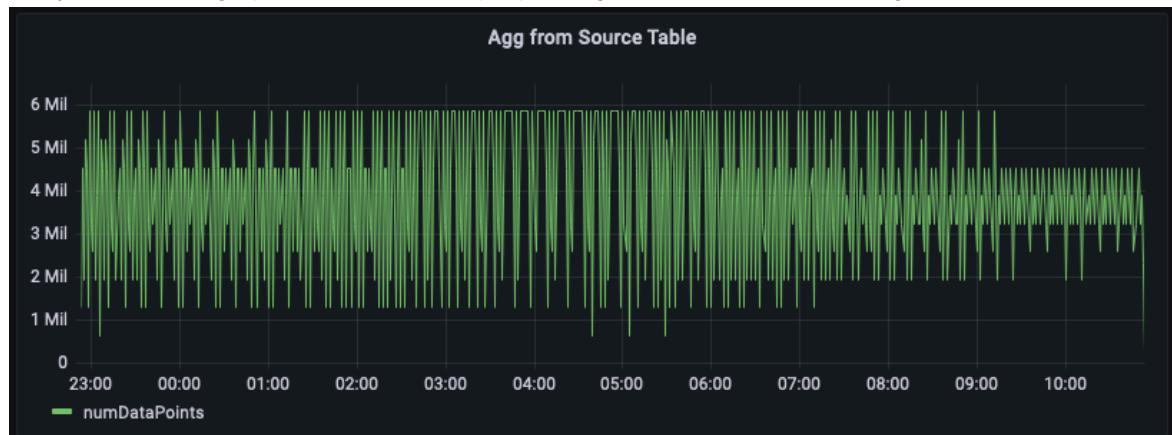
- How to take your dashboard query that is used to obtain aggregate statistics and map it to a scheduled query.
- How Timestream manages the execution of the different instances of your scheduled query.
- How you can have different instances of scheduled queries overlap in time ranges and how the correctness 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 \(p. 182\)](#)
- [Scheduled query to pre-compute aggregates \(p. 183\)](#)
- [Aggregate from derived table \(p. 185\)](#)
- [Aggregate combining source and derived tables \(p. 186\)](#)
- [Aggregate from frequently refreshed scheduled computation \(p. 187\)](#)

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 allows you to materialize these pre-computations in another Timestream 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 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. 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 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

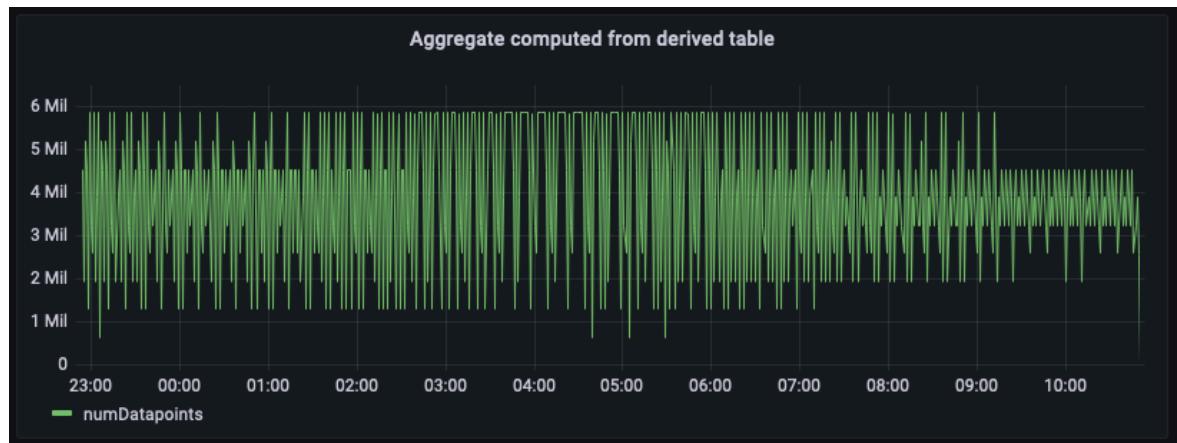
considerations are discussed below on how you trade-off this refresh cadence with freshness of the data and how you address updating the aggregates for data that arrives even more delayed or if your source of the scheduled computation has updated values which would require the aggregates to be recomputed.

Every scheduled computation has a notification configuration where Timestream 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, i.e., 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 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.

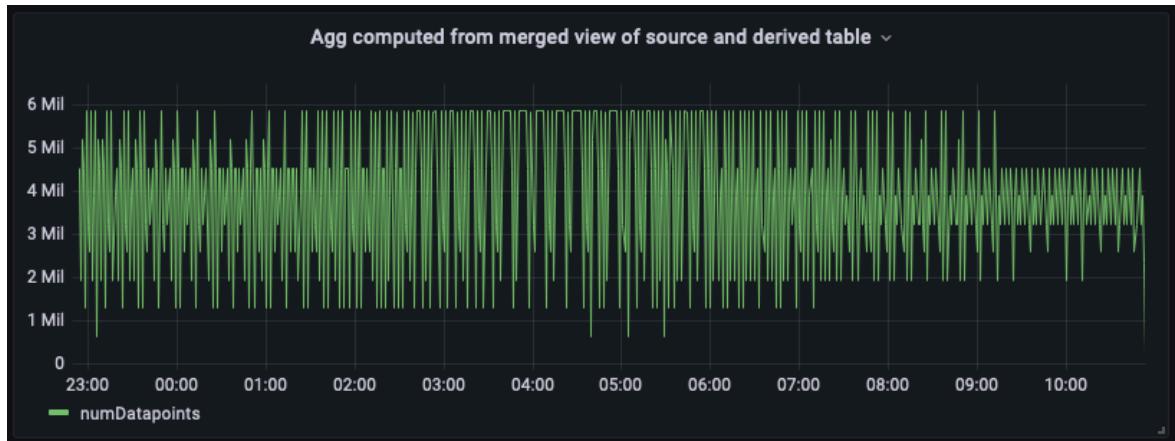
Aggregate combining source and derived tables

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'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 expression `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": "*****"
}

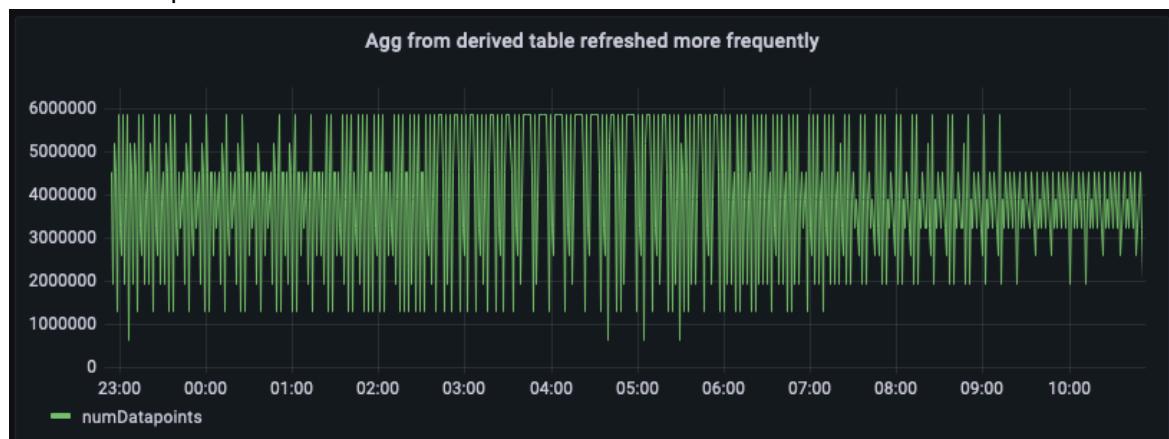
```

```

SELECT bin(time, 1m) as minute, SUM(numDataPoints) as numDatapoints
FROM "derived"."per_minute_aggs_pt1m"
WHERE time BETWEEN from_milliseconds(163669996446) 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 \(p. 189\)](#)
- [Derived table to precompute at daily granularity \(p. 189\)](#)
- [Computed from derived table \(p. 191\)](#)
- [Combining from source and derived table \(p. 191\)](#)

Computed from source table

Below is an example query to find the last measurement emitted by the services in a specific deployment (i.e., 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.

```
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 scheduled computation to pre-compute the values for your entire fleet. That is the power of Timestream'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 issued 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 \(p. 192\)](#)
- [Pre-compute unique dimension values \(p. 192\)](#)
- [Computing the variables from derived table \(p. 194\)](#)

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"
        }
      ]
    }
  }
}
```

```

        "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, i.e., the time when the data was ingested into Timestream 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 arrival data. Below 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 \(p. 195\)](#)
- [Manual executions for unpredictable late arriving data \(p. 198\)](#)

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'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 arrival 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 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.

```
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 04:00:00, 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 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 the 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 automatically manages the schedules so that you don't see these issues even if the subsequent invocations have overlapping time ranges.
- As 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 manages executions of the queries moving forward where the refresh is governed by the schedule expression you provide. Depending on 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'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 allow you to express your queries using the full SQL surface area of Timestream. Your query can include one or more source tables, perform aggregations or any other query allowed by Timestream's SQL language, and then store the results of the query in another destination table in Timestream.

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. 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 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.

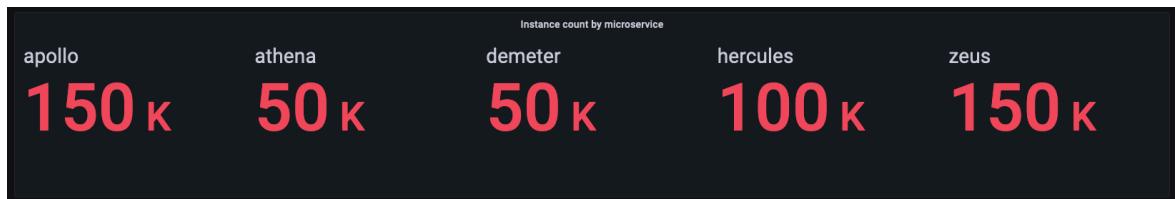
Topics

- [Converting an aggregate dashboard to scheduled query \(p. 201\)](#)
- [Using scheduled queries and raw data for drill downs \(p. 203\)](#)
- [Optimizing costs by sharing scheduled query across dashboards \(p. 207\)](#)

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",

```

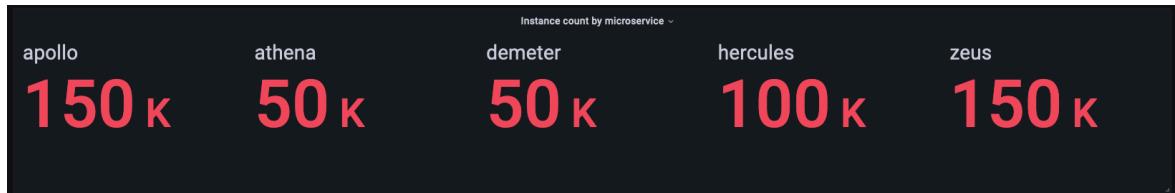
```

        "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

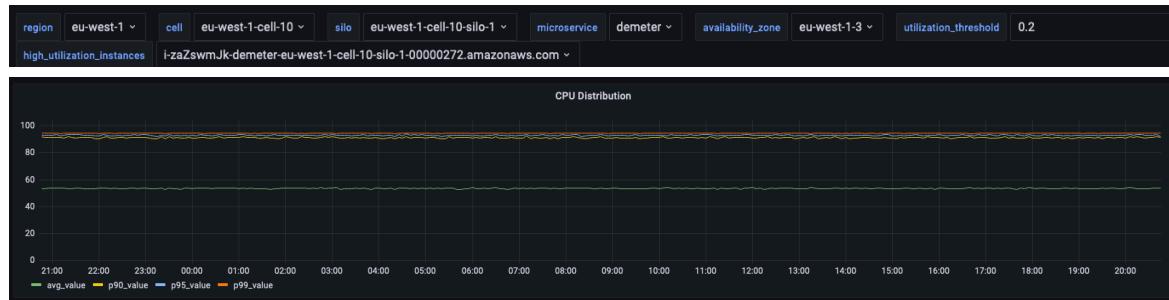
You can use the aggregated statistics across your fleet to identify areas that need drill downs and then use the raw data to drill down into granular data to get deeper insights.

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.

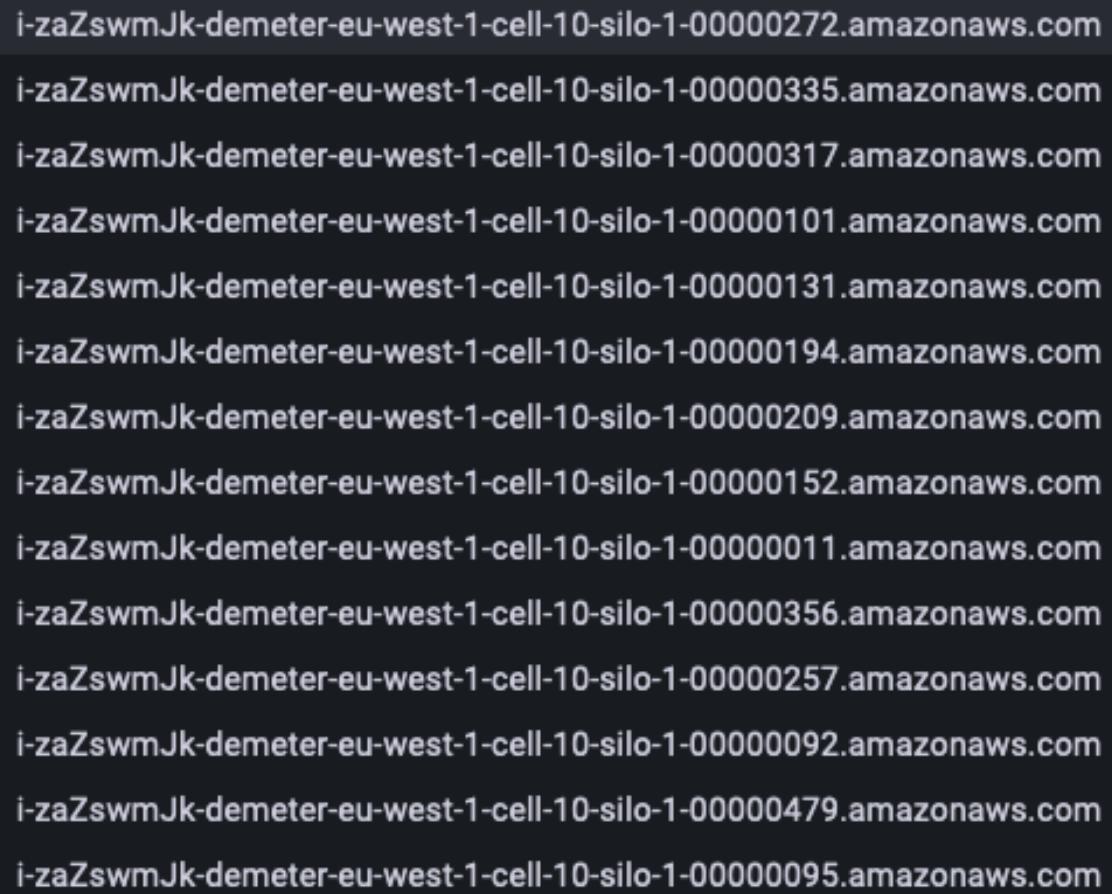
```
WITH microservice_cell_avg AS (
  SELECT AVG(cpu_user) AS microservice_avg_metric
  FROM "raw_data"."devops"
 WHERE $__timeFilter
   AND measure_name = 'metrics'
   AND region = '${region}'
   AND cell = '${cell}'
   AND silo = '${silo}'
   AND availability_zone = '${availability_zone}'
   AND microservice_name = '${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.



The screenshot shows a dropdown menu with a list of instance names. The first item is highlighted in a darker shade, indicating it is the currently selected instance. The list contains 15 entries, each representing a unique instance identifier.

- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000272.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000335.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000317.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000101.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000131.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000194.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000209.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000152.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000011.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000356.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000257.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000092.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000479.amazonaws.com
- i-zaZswmJk-demeter-eu-west-1-cell-10-silo-1-00000095.amazonaws.com



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 one.

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's SQL support to perform complex analytical tasks with common table expressions, window functions, joins, and so on.

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

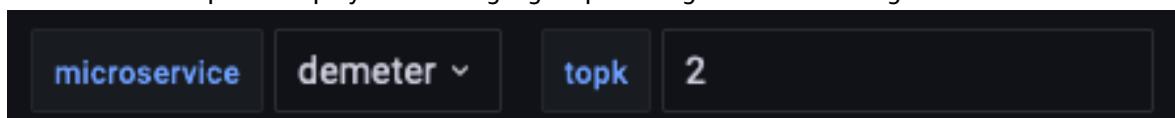
```

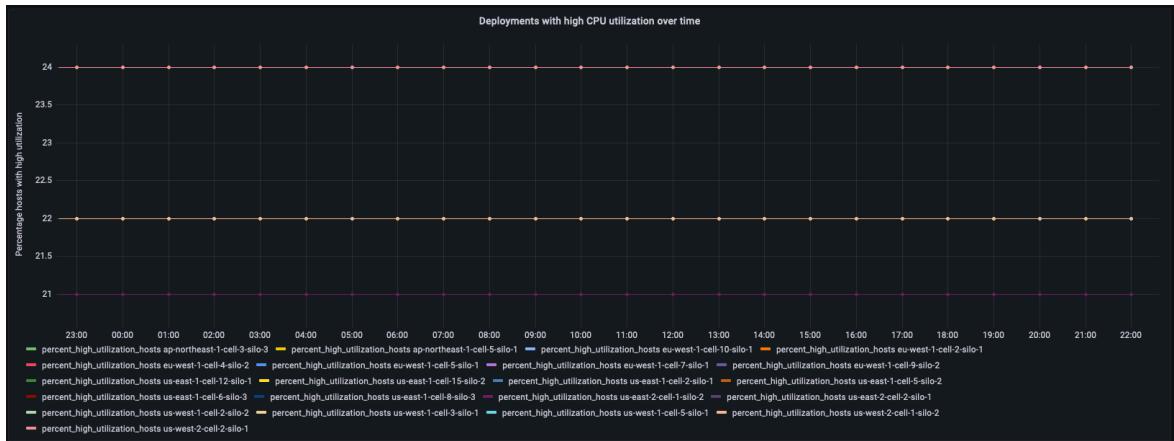
Drill down into a microservice to find hot spots

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 whose 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 use 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 deployments

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
```

Adding tags and labels to resources

You can label Amazon Timestream 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, 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 \(p. 215\)](#).
2. Create tags by using [Tagging operations \(p. 215\)](#).

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 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 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 console, query language, or the AWS Command Line Interface (AWS CLI).

For bulk editing, you can also use Tag Editor on the AWS Management Console. For more information, see [Working with Tag Editor](#) in the *AWS Resource Groups User Guide*.

Topics

- [Adding tags to new or existing databases and tables using the console \(p. 216\)](#)

Adding tags to new or existing databases and tables using the console

You can use the Timestream 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 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 \(p. 215\)](#).

Security in Timestream

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, 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. The following topics show you how to configure Timestream 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 resources.

Topics

- [Data protection in Timestream \(p. 217\)](#)
- [Identity and access management for Amazon Timestream \(p. 219\)](#)
- [Logging and monitoring in Timestream \(p. 249\)](#)
- [Resilience in Amazon Timestream \(p. 258\)](#)
- [Infrastructure security in Amazon Timestream \(p. 259\)](#)
- [Configuration and vulnerability analysis in Timestream \(p. 259\)](#)
- [Incident response in Timestream \(p. 259\)](#)
- [VPC endpoints \(AWS PrivateLink\) \(p. 259\)](#)
- [Security best practices for Amazon Timestream \(p. 262\)](#)

Data protection in Timestream

The AWS [shared responsibility model](#) applies to data protection in Amazon Timestream. 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. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the [AWS Security Blog](#).

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with 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 recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- 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 personal data that is stored in Amazon S3.
- If you require FIPS 140-2 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 \(FIPS\) 140-2](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form fields such as a **Name** field. This includes when you work with Timestream or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form 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 data protection topics like Encryption at Rest and Key Management, select any of the available topics below.

Topics

- [Encryption at rest \(p. 218\)](#)
- [Encryption in transit \(p. 218\)](#)
- [Key management \(p. 219\)](#)
- [Internetwork traffic privacy \(p. 219\)](#)

Encryption at rest

Timestream 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 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.
- 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 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 data. For more information on keys in Timestream, see [Key management \(p. 219\)](#).

All the data under any of your Timestream databases is encrypted using 1 KMS Customer Managed Key (CMK). Timestream stores your data in two storage tiers, memory store and magnetic store. Memory store data is encrypted using a Timestream service key. Magnetic store data is encrypted using your KMS CMK.

The Timestream Query service requires credentials to access your data. These credentials are encrypted using your KMS key.

Encryption in transit

All your Timestream data is encrypted in transit. By default, all communications to and from Timestream are protected by using Transport Layer Security (TLS) encryption.

See [Internetwork traffic privacy \(p. 219\)](#) for more information.

Key management

You can manage keys for Amazon Timestream using the [AWS Key Management Service \(AWS KMS\)](#). **Timestream 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-managed key*: If you do not provide a key, Timestream will create a alias/aws/timestream key using KMS.
- *Customer-managed key (CMK)*: KMS CMK's 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-owned key*: If you do not provide a key, Timestream 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 (CMK)*: KMS CMK's 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.

Internetwork traffic privacy

All traffic to and from Amazon Timestream is secured using TLS 1.2.

Timestream does not initiate cross region traffic between services.

Identity and access management for Amazon Timestream

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 resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience \(p. 220\)](#)
- [Authenticating with identities \(p. 220\)](#)
- [Managing access using policies \(p. 222\)](#)
- [How Amazon Timestream works with IAM \(p. 223\)](#)
- [AWS managed policies for Amazon Timestream \(p. 227\)](#)
- [Amazon Timestream identity-based policy examples \(p. 235\)](#)
- [Troubleshooting Amazon Timestream identity and access \(p. 247\)](#)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Timestream.

Service user – If you use the Timestream service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Timestream features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Timestream, see [Troubleshooting Amazon Timestream identity and access \(p. 247\)](#).

Service administrator – If you're in charge of Timestream resources at your company, you probably have full access to Timestream. It's your job to determine which Timestream features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Timestream, see [How Amazon Timestream works with IAM \(p. 223\)](#).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Timestream. To view example Timestream identity-based policies that you can use in IAM, see [Amazon Timestream identity-based policy examples \(p. 235\)](#).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (successor to AWS Single Sign-On) (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account in the AWS Sign-In User Guide](#).

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [Signature Version 4 signing process](#) in the [AWS General Reference](#).

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the [AWS IAM Identity Center \(successor to AWS Single Sign-On\) User Guide](#) and [Using multi-factor authentication \(MFA\) in AWS](#) in the [IAM User Guide](#).

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the *AWS account root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the [AWS General Reference](#).

IAM users and groups

An [IAM user](#) is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the [IAM User Guide](#).

An [IAM group](#) is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the [IAM User Guide](#).

IAM roles

An [IAM role](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the [IAM User Guide](#).

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Creating a role for a third-party Identity Provider](#) in the [IAM User Guide](#). If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see [Permission sets](#) in the [AWS IAM Identity Center \(successor to AWS Single Sign-On\) User Guide](#).
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the [IAM User Guide](#).
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
 - **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see [Actions, Resources, and Condition Keys for Amazon Timestream](#) in the [Service Authorization Reference](#).
 - **Service role** – 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 [Creating a role to delegate permissions to an AWS service](#) in the [IAM User Guide](#).

- **Service-linked role** – 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 IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) 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 is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

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**.

Every IAM entity (user or role) starts with no permissions. By default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

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 [Creating IAM policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing 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 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.

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, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. 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 works with IAM

Before you use IAM to manage access to Timestream, you should understand what IAM features are available to use with Timestream. To get a high-level view of how Timestream and other AWS services work with IAM, see [AWS Services That Work with IAM](#) in the *IAM User Guide*.

Topics

- [Timestream identity-based policies \(p. 224\)](#)
- [Timestream resource-based policies \(p. 226\)](#)

- [Authorization based on Timestream tags \(p. 226\)](#)
- [Timestream IAM roles \(p. 227\)](#)

Timestream 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 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. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

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 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 (p. 225)	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 (p. 225) .	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 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 can run queries without resources. Now, consider a Select query:

```
SELECT * FROM database.table
```

This type of query requires a resource, specifically an Timestream 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. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its [Amazon Resource Name \(ARN\)](#). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

In Timestream databases and tables can be used in the Resource element of IAM permissions.

The Timestream database resource has the following ARN:

```
arn:${Partition}:timestream:${Region}:${Account}:database/${DatabaseName}
```

The Timestream 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 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 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 identity-based policies, see [Amazon Timestream identity-based policy examples](#) (p. 235).

Timestream resource-based policies

Timestream 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 tags

You can manage access to your Timestream 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 resources, see [Tagging resources \(p. 215\)](#).

To view example identity-based policies for limiting access to a resource based on the tags on that resource, see [Timestream resource access based on tags \(p. 242\)](#).

Timestream IAM roles

An [IAM role](#) is an entity within your AWS account that has specific permissions.

Using temporary credentials with Timestream

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 does not support service-linked roles.

Service roles

Timestream does not support service roles.

AWS managed policies for Amazon Timestream

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*.

Timestream updates to AWS managed policies

View details about updates to AWS managed policies for Timestream since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the [Timestream Document history page](#).

Change	Description	Date
AmazonTimestreamReadOnlyAccess – Update to an existing policy	<p>AmazonTimestreamReadOnlyAccess policy provides read-only access to Amazon Timestream and permission to cancel any running query.</p> <p>Timestream added the <code>timestream:DescribeScheduledQuery</code> and <code>timestream>ListScheduledQueries</code> actions to the existing managed policy <code>AmazonTimestreamReadOnlyAccess</code>. These actions are used when listing and describing existing scheduled queries.</p> <p>The updated <code>AmazonTimestreamReadOnlyAccess</code> policy can be viewed below. The policy update does not change the effect or usage of either managed policies.</p> <pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["timestream:CancelQuery", "timestream:DescribeDatabase", "timestream:DescribeEndpoints", "timestream:DescribeTable", "timestream>ListDatabases", "timestream>ListMeasures", "timestream>ListTables", "timestream>ListTagsForResource", "timestream:Select", "timestream:SelectValues", "timestream:DescribeScheduledQuery", "timestream>ListScheduledQueries"], "Resource": "*" }] }</pre>	November 29, 2021

Change	Description	Date
] }	

Change	Description	Date
AmazonTimestreamConsoleFullAccess Update to an existing policy	<p>AmazonTimestreamConsoleFullAccess provides full access to manage Amazon Timestream using the Amazon Management Console. This policy also grants permissions for certain KMS operations, S3 operations, and operations to manage your saved queries.</p> <p>Timestream added the s3>ListAllMyBuckets action to the existing managed policy AmazonTimestreamConsoleFullAccess. This action is used when you specify an S3 bucket for Timestream to log magnetic store write errors.</p> <p>The updated AmazonTimestreamConsoleFullAccess policy can be viewed below. The policy update does not change the effect or usage of the managed policy.</p> <pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["timestream:*"], "Resource": "*" }, { "Effect": "Allow", "Action": ["kms:DescribeKey", "kms>ListKeys", "kms>ListAliases"], "Resource": "*" }, { "Effect": "Allow", "Action": ["kms>CreateGrant"], "Resource": "*", "Condition": { "StringLike": { "kms:Principal": "arn:aws:timestream:region:account-id:stream/*" } } }] }</pre>	November 29, 2021

Change	Description	Date
	<pre> "Condition": { "ForAnyValue:StringEquals": { "kms:EncryptionContextKeys": "aws:timestream:database-name" }, "Bool": { "kms:GrantIsForAWSResource": true }, "StringLike": { "kms:ViaService": "timestream.*.amazonaws.com" } }, { "Effect": "Allow", "Action": ["dbqms>CreateFavoriteQuery", "dbqms>DescribeFavoriteQueries", "dbqms>UpdateFavoriteQuery", "dbqms>DeleteFavoriteQueries", "dbqms>GetQueryString", "dbqms>CreateQueryHistory", "dbqms>DescribeQueryHistory", "dbqms>UpdateQueryHistory", "dbqms>DeleteQueryHistory"], "Resource": "*" }, { "Effect": "Allow", "Action": ["s3>ListAllMyBuckets"], "Resource": "*" } } </pre>	

Change	Description	Date
AmazonTimestreamFullAccess – Update to an existing policy	<p>AmazonTimestreamFullAccess provides administrative access to Amazon Timestream. Note that this policy also grants permissions for certain KMS and S3 operations.</p> <p>Timestream added the <code>s3>ListAllMyBuckets</code> action to the existing managed policy <code>AmazonTimestreamFullAccess</code>. This action is used when you specify an S3 bucket for Timestream to log magnetic store write errors.</p> <p>The updated <code>AmazonTimestreamFullAccess</code> policy can be viewed below. The policy update does not change the effect or usage of the policy.</p> <pre>{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["timestream:*"], "Resource": "*" }, { "Effect": "Allow", "Action": ["kms:DescribeKey"], "Resource": "*" }, { "Effect": "Allow", "Action": ["kms>CreateGrant"], "Resource": "*", "Condition": { "ForAnyValue:StringEquals": { "kms:EncryptionContextKeys": "aws:timestream:database-name" } } }] }</pre>	November 29, 2021

Change	Description	Date
	<pre> "Bool": { "kms:GrantIsForAWSResource": true }, "StringLike": { "kms:ViaService": "timestream.*.amazonaws.com" } }, { "Effect": "Allow", "Action": ["s3>ListAllMyBuckets"], "Resource": "*" } } }</pre>	

Change	Description	Date
AmazonTimestreamConsoleFullAccess – Update to an existing policy	<p>Timestream removed redundant actions from the existing managed policy AmazonTimestreamConsoleFullAccess (p. 19). Previously, the AmazonTimestreamConsoleFullAccess managed policy included a redundant action, <i>dbqms:DescribeQueryHistory</i>. The old policy can be viewed below:</p> <pre> "Action": ["dbqms:CreateFavoriteQuery", "dbqms:DescribeFavoriteQueries", "dbqms:UpdateFavoriteQuery", "dbqms:DeleteFavoriteQueries", "dbqms:GetQueryString", "dbqms:CreateQueryHistory", "dbqms:DescribeQueryHistory", "dbqms:UpdateQueryHistory", "dbqms:DeleteQueryHistory", "dbqms:DescribeQueryHistory"]</pre> <p>The updated policy removes the redundant action, and can be viewed below. The policy update does not change the effect or usage of the AmazonTimestreamConsoleFullAccess managed policy.</p> <pre> "Action": ["dbqms:CreateFavoriteQuery", "dbqms:DescribeFavoriteQueries", "dbqms:UpdateFavoriteQuery", "dbqms:DeleteFavoriteQueries", "dbqms:GetQueryString", "dbqms:CreateQueryHistory", "dbqms:UpdateQueryHistory",]</pre>	April 23, 2021

Change	Description	Date
	<pre>"dbqms:DeleteQueryHistory", "dbqms:DescribeQueryHistory"]</pre>	
Timestream started tracking changes	Timestream started tracking changes for its AWS managed policies.	April 21, 2021

Amazon Timestream identity-based policy examples

By default, IAM users and roles don't have permission to create or modify Timestream 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 \(p. 235\)](#)
- [Using the Timestream console \(p. 236\)](#)
- [Allow users to view their own permissions \(p. 236\)](#)
- [Common operations in Timestream \(p. 237\)](#)
- [Timestream resource access based on tags \(p. 242\)](#)
- [Scheduled queries \(p. 244\)](#)

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Timestream 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 AWS 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 [IAM Access Analyzer policy validation](#) in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or root users in your 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 [Configuring MFA-protected API access](#) 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 console

Timestream does not require specific permissions to access the Amazon Timestream console. You need at least read-only permissions to list and view details about the Timestream 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

Below are sample IAM policies that allow for common operations in the Timestream service.

Topics

- [Allowing all operations \(p. 237\)](#)
- [Allowing SELECT operations \(p. 237\)](#)
- [Allowing SELECT operations on multiple resources \(p. 238\)](#)
- [Allowing metadata operations \(p. 238\)](#)
- [Allowing INSERT operations \(p. 239\)](#)
- [Allowing CRUD operations \(p. 239\)](#)
- [Cancel queries and select data without specifying resources \(p. 240\)](#)
- [Create, describe, delete and describe a database \(p. 240\)](#)
- [Limit listed databases by tag{"Owner": "\\${username}"} \(p. 240\)](#)
- [List all tables in a database \(p. 241\)](#)
- [Create, describe, delete, update and select on a table \(p. 241\)](#)
- [Limit a query by table \(p. 242\)](#)

Allowing all operations

The following is a sample policy that allows all operations in Timestream.

```
{  
  "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.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "timestream:Select",  
        "timestream:DescribeTable",  
        "timestream>ListMeasures"  
      ],  
      "Resource": "arn:aws:timestream:us-east-1:<account_ID>: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.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:Select",
                "timestream:DescribeTable",
                "timestream>ListMeasures"
            ],
            "Resource": [
                "arn:aws:timestream:us-east-1:<account_ID>:database/sampleDB/
table/DevOps",
                "arn:aws:timestream:us-east-1:<account_ID>:database/sampleDB/
table/DevOps1",
                "arn:aws:timestream:us-east-1:<account_ID>: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.

```

{
    "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.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "timestream:WriteRecords"
            ],
            "Resource": [
                "arn:aws:timestream:us-east-1:<account_id>: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.

```

{
    "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:

```
{
    "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:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream>CreateDatabase",
                "timestream>DescribeDatabase",
                "timestream>DeleteDatabase",
                "timestream>UpdateDatabase"
            ],
            "Resource": "arn:aws:timestream:us-east-1:<account_ID>: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}"}:

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "timestream>ListDatabases"  
      ],  
      "Resource": "arn:aws:timestream:us-east-1:<account_ID>: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:

```
{  
  {  
    "Version": "2012-10-17",  
    "Statement": [  
      {  
        "Effect": "Allow",  
        "Action": [  
          "timestream>ListTables"  
        ],  
        "Resource": "arn:aws:timestream:us-east-1:<account_ID>: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:

```
{  
  "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:<account_ID>: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:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "timestream:Select"
            ],
            "Resource": "arn:aws:timestream:us-east-1:<account_ID>:database/
sampleDB/table/*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "timestream:Select"
            ],
            "Resource": "arn:aws:timestream:us-east-1:<account_ID>:database/
sampleDB/table/DevOps"
        }
    ]
}
```

Timestream resource access based on tags

You can use conditions in your identity-based policy to control access to Timestream 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.

```
{
    "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 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`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CreateTagTableUser",
      "Effect": "Allow",
      "Action": ["timestream:Create", "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`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowDescribeEndpoints",
      "Effect": "Allow",
      "Action": [
        "timestream:DescribeEndpoints"
      ],
      "Resource": "*"
    },
    {
      "Sid": "AllowDevTestAccess",
      "Effect": "Allow",
      "Action": [
        "timestream:DescribeDatabase"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "timestream:tag/env": [
            "dev",
            "test"
          ]
        }
      }
    }
  ]
}
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "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.

```

{
    "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>*.

```

{
    "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>*.

```

{
    "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.

```

{
    "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.

```
{
  "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*.

```
{  
  "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:Account_ID: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`.

```
{  
  "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:Account_ID:scheduled-  
query/Scheduled_Query_Name*"  
        }  
      }  
    }  
  ]  
}
```

Troubleshooting Amazon Timestream identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Timestream and IAM.

Topics

- [I Am not authorized to perform an action in Timestream \(p. 248\)](#)
- [I Am not authorized to perform iam:PassRole \(p. 248\)](#)

- [I want to view my access keys \(p. 248\)](#)
- [I'm an administrator and want to allow others to access Timestream \(p. 249\)](#)
- [I want to allow people outside of my AWS account to access my Timestream resources \(p. 249\)](#)

I Am not authorized to perform an action in Timestream

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` 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.

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. 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 view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

Important

Do not provide your access keys to a third party, even to help [find your canonical user ID](#). By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys.

If you already have two, you must delete one key pair before creating a new one. To view instructions, see [Managing access keys](#) in the *IAM User Guide*.

I'm an administrator and want to allow others to access Timestream

To allow others to access Timestream, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in Timestream.

To get started right away, see [Creating your first IAM delegated user and group](#) in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my Timestream 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 supports these features, see [How Amazon Timestream works with IAM \(p. 223\)](#).
- 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*.

Logging and monitoring in Timestream

Monitoring is an important part of maintaining the reliability, availability, and performance of Timestream 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, 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 performance in your environment, by measuring performance at various times and under different load conditions. As you monitor

Timestream, 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 \(p. 250\)](#)
- [Monitoring with Amazon CloudWatch \(p. 251\)](#)
- [Logging Timestream API calls with AWS CloudTrail \(p. 257\)](#)

Monitoring tools

AWS provides various tools that you can use to monitor Timestream. 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 \(p. 250\)](#)
- [Manual monitoring tools \(p. 250\)](#)

Automated monitoring tools

You can use the following automated monitoring tools to watch Timestream 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 \(p. 251\)](#).

Manual monitoring tools

Another important part of monitoring Timestream involves manually monitoring those items that the CloudWatch alarms don't cover. The Timestream, 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

Monitoring with Amazon CloudWatch

You can monitor Timestream using Amazon CloudWatch, which collects and processes raw data from Timestream 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 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 metrics? \(p. 251\)](#)
- [Timestream metrics and dimensions \(p. 251\)](#)
- [Creating CloudWatch alarms to monitor Timestream \(p. 257\)](#)

How do I use Timestream metrics?

The metrics reported by Timestream 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 <code>SystemErrors</code> 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 investigate.

Timestream metrics and dimensions

When you interact with Timestream, 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.

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.

To view metrics using the AWS CLI

- At a command prompt, use the following command.

```
aws cloudwatch list-metrics --namespace "AWS/Timestream"
```

Timestream metrics and dimensions

The following CloudWatch metrics are available in Timestream:

- `SuccessfulRequestLatency`
- `SystemErrors`
- `UserErrors`
- `CumulativeBytesMetered`

The following dimensions are available:

- `Operation`
- `DatabaseName`
- `TableName`

Topics

- [Timestream metrics \(p. 252\)](#)
- [Dimensions for Timestream metrics \(p. 256\)](#)

Timestream metrics

Amazon CloudWatch aggregates the following Timestream metrics:

Note

All metrics are aggregated at one-minute intervals.

Metric	Description
<code>SystemErrors</code>	<p>The requests to Timestream that generate a <code>SystemError</code> during the specified time period. A <code>SystemError</code> usually indicates an internal service error.</p> <p>Units: Count</p> <p>Dimensions: <code>Operation</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• <code>Sum</code>• <code>SampleCount</code>
<code>UserErrors</code>	<p>Requests to Timestream that generate an <code>InvalidRequest</code> error during the specified time period. An <code>InvalidRequest</code> 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.</p> <p><code>UserErrors</code> represents the aggregate of invalid requests for the current AWS Region and the current AWS account.</p> <p>Units: Count</p> <p>Dimensions: <code>Operation</code></p>

Metric	Description
	<p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount
SuccessfulRequestLatency	<p>The successful requests to Timestream during the specified time period. SuccessfulRequestLatency can provide two different kinds of information:</p> <ul style="list-style-type: none"> • The elapsed time for successful requests (Minimum, Maximum, Sum, or Average). • The number of successful requests (SampleCount). <p>SuccessfulRequestLatency reflects activity only within Timestream and does not take into account network latency or client-side activity.</p> <p>Units: Milliseconds</p> <p>Dimensions</p> <ul style="list-style-type: none"> • DatabaseName • TableName • Operation <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • P10 • p50 • p90 • p95 • p99
CumulativeBytesMetered	<p>The amount of data scanned by queries sent to Timestream, in bytes.</p> <p>Units: Bytes</p> <p>Dimensions: Operation</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum

Metric	Description
MagneticStoreRejectedRecordCount	<p>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.</p> <p>Units: Count</p> <p>Dimensions</p> <ul style="list-style-type: none"> • DatabaseName • TableName • Operation <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount
MagneticStoreRejectedUploadUserFailures	<p>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.</p> <p>Units: Count</p> <p>Dimensions</p> <ul style="list-style-type: none"> • DatabaseName • TableName • Operation <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount

Metric	Description
MagneticStoreRejectedUploadSystemFailure	<p>The number of magnetic store rejected record reports that were not uploaded due to system errors.</p> <p>Units: Count</p> <p>Dimensions</p> <ul style="list-style-type: none"> • DatabaseName • TableName • Operation <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount
ActiveMagneticStorePartitions	<p>The number of magnetic store partitions actively ingesting data at a given time.</p> <p>Units: Count</p> <p>Dimensions</p> <ul style="list-style-type: none"> • DatabaseName • Operation <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount

Metric	Description
MagneticStorePendingRecordsLatency	<p>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.</p> <p>Units: Milliseconds</p> <p>Dimensions</p> <ul style="list-style-type: none"> • DatabaseName • TableName • Operation <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • P10 • p50 • p90 • p95 • p99

Important

Not all statistics, such as Average or Sum, are applicable for every metric. However, all of these values are available through the Timestream console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics.

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 table-level operations have DatabaseName and TableName as dimensions, but account level operations do not. See [Dimensions for Timestream metrics \(p. 256\)](#).

[Dimensions for Timestream metrics](#)

The metrics for Timestream are qualified by the values for the account, table name, or operation. You can use the CloudWatch console to retrieve Timestream data along any of the dimensions in the following table:

Dimension	Description
DatabaseName	This dimension limits the data to a specific Timestream database. This value can be any database in the current Region and the current AWS account

Dimension	Description
Operation	This dimension limits the data to one of the Timestream operations. See the Timestream Query API Reference for a list of available values.
TableName	This dimension limits the data to a specific table in a Timestreams database.

Creating CloudWatch alarms to monitor Timestream

You can create an Amazon CloudWatch alarm for Timestream 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](#).

Logging Timestream API calls with AWS CloudTrail

Timestream is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Timestream. CloudTrail captures Data Definition Language (DDL) API calls for Timestream as events. The calls that are captured include calls from the Timestream console and code calls to the Timestream 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. 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, 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 information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Timestream, 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 generates CloudTrail Events for management APIs, but does not generate events for data plane APIs. Specifically, Timestream DOES NOT emit events for:

- *WriteRecords*
- *Query*

Note

When using the Timestream Query API via a query string, none of the supported SQL constructs are logged by CloudTrail. Timestream's supported SQL constructs map to the following Timestream IAM actions, as follows.

Query string	IAM action
Select	Select, SelectValues
Show measures	ListMeasures
Describe Database	DescribeDatabase
Describe Table	DescribeTable
Show Databases	ListDatabases
Show Tables	ListTables

However, direct API calls to `DescribeDatabase`, `DescribeTable`, `ListDatabases` and `ListTables` using the Timestream SDK will be logged to CloudTrail.

- *DescribeEndpoints*

For an ongoing record of events in your AWS account, including events for Timestream, 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](#)

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](#).

Resilience in Amazon Timestream

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](#).

While Amazon Timestream is multi-AZ, it does not support backups to other AWS Availability Zones or Regions. However, you can write your own application using the Timestream SDK to query data and save it to the destination of your choice.

Infrastructure security in Amazon Timestream

As a managed service, Amazon Timestream 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 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 is architected so that your traffic is isolated to the specific AWS Region that your Timestream 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 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

Amazon Timestream service incidents are reported in the [Personal Health Dashboard](#). You can learn more about the dashboard and AWS Health [here](#).

Timestream supports reporting using AWS CloudTrail. For more information, see [Logging Timestream API calls with AWS CloudTrail \(p. 257\)](#).

VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon Timestream by creating an *interface VPC endpoint*. Interface endpoints are powered by [AWS PrivateLink](#), a technology that enables you to privately access Timestream 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 APIs. Traffic between your VPC and Timestream 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 and VPC endpoints, we've provided information on specific considerations for Timestream with VPC endpoints, creating an interface VPC endpoint for Timestream, creating a VPC endpoint policy for Timestream, and using the Timestream client (for either the Write or Query SDK) with VPC endpoints..

Topics

- [How VPC endpoints work with Timestream \(p. 260\)](#)
- [Creating an interface VPC endpoint for Timestream \(p. 260\)](#)
- [Creating a VPC endpoint policy for Timestream \(p. 262\)](#)

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 \(p. 6\)](#). For example, suppose that your account has been mapped to cell11 in us-west-2, and you've set up VPC interface endpoints for writes (`ingest-cell11.timestream.us-west-2.amazonaws.com`) and queries (`query-cell11.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, ensure that you review [Interface endpoint properties and limitations](#) in the [Amazon VPC User Guide](#).
- Timestream supports making calls to [all of its API actions](#) from your VPC.
- VPC endpoint policies are supported for Timestream. By default, full access to Timestream 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 \(p. 260\)](#) section of this guide.

Now that you understand how Timestream works with VPC endpoints, [create an interface VPC endpoint for Timestream \(p. 260\)](#).

Creating an interface VPC endpoint for Timestream

You can create an [interface VPC endpoint](#) for the Timestream 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. \(p. 260\)](#)

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 \(p. 261\)](#) 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 \(p. 261\)](#) 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, [create a VPC endpoint policy for Timestream \(p. 262\)](#).

Creating a VPC endpoint policy for Timestream

You can attach an endpoint policy to your VPC endpoint that controls access to Timestream. 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 actions

The following is an example of an endpoint policy for Timestream. When attached to an endpoint, this policy grants access to the listed Timestream 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

Amazon Timestream 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 preventative security best practices \(p. 262\)](#)

Timestream preventative security best practices

The following best practices can help you anticipate and prevent security incidents in Timestream.

Encryption at rest

Timestream 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 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 Encryption at Rest](#).

Use IAM roles to authenticate access to Timestream

For users, applications, and other AWS services to access Timestream, 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 base authorization

When granting permissions, you decide who is getting them, which Timestream 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 resources.

You can do this by using the following:

- [AWS managed \(predefined\) policies](#)
- [Customer managed policies](#)
- [Tag-based authorization \(p. 226\)](#)

Consider client-side encryption

If you store sensitive or confidential data in Timestream, 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 integrates with a variety of AWS services and popular third-party tools. Currently, Timestream supports integrations with the following:

Topics

- [AWS Lambda \(p. 264\)](#)
- [AWS IoT Core \(p. 265\)](#)
- [Amazon Kinesis Data Analytics for Apache flink \(p. 268\)](#)
- [Amazon Kinesis \(p. 269\)](#)
- [Amazon MSK \(p. 269\)](#)
- [Amazon QuickSight \(p. 270\)](#)
- [Amazon SageMaker \(p. 272\)](#)
- [Grafana \(p. 273\)](#)
- [Open source Telegraf \(p. 275\)](#)
- [JDBC \(p. 278\)](#)
- [VPC endpoints \(AWS PrivateLink\) \(p. 288\)](#)

AWS Lambda

You can create Lambda functions that interact with Timestream. 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 with Python \(p. 264\)](#)
- [Build AWS Lambda functions using Amazon Timestream with JavaScript \(p. 265\)](#)
- [Build AWS Lambda functions using Amazon Timestream with Go \(p. 265\)](#)
- [Build AWS Lambda functions using Amazon Timestream with C# \(p. 265\)](#)

Build AWS Lambda functions using Amazon Timestream with Python

To build AWS Lambda functions using Amazon Timestream 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 [Create an IAM user with Timestream access \(p. 19\)](#).
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 with JavaScript

To build AWS Lambda functions using Amazon Timestream 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 with Go

To build AWS Lambda functions using Amazon Timestream with Go, follow the instructions outlined [here](#).

Related references are at [timestreamwrite](#) and [timestreamquery](#).

Build AWS Lambda functions using Amazon Timestream with C#

To build AWS Lambda functions using Amazon Timestream 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 \(p. 266\)](#)
- [Using the console \(p. 267\)](#)
- [Using the CLI \(p. 267\)](#)
- [Sample application \(p. 268\)](#)
- [Video tutorial \(p. 268\)](#)

Prerequisites

1. Create a database in Amazon Timestream using the instructions described in [Create a database \(p. 23\)](#).

2. Create a table in Amazon Timestream using the instructions described in [Create a table \(p. 23\)](#).

Using the console

1. Use the AWS Management Console for AWS IoT Core to create a rule by clicking on Act, then Rule, then Create.
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 Kinesis Data Analytics for Apache flink

You can use Apache Flink to transfer your time series data from Amazon Kinesis Data Analytics, Amazon MSK, Apache Kafka, and other streaming technologies directly into Amazon Timestream. 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 Kinesis Data Analytics, 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 Kinesis Data Analytics 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 \(p. 269\)](#)

- [Video tutorial \(p. 269\)](#)

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 \(p. 23\)](#)
2. Create a table in Timestream with the name kinesisdata1 following the instructions described in [Create a table \(p. 23\)](#)
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 Kinesis Data Analytics application following the instructions for [Compiling the Application Code](#)
7. Upload the Kinesis Data Analytics 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 Kinesis Data Analytics to Timestream following the instructions to [Create a table \(p. 23\)](#)

Video tutorial

This [video](#) explains how to use Timestream with Kinesis Data Analytics for Apache Flink.

Amazon Kinesis

You can send data from Amazon Kinesis to Amazon Timestream using the sample Timestream data connector for Kinesis Data Analytics for Apache Flink. Refer to [Amazon Kinesis Data Analytics for Apache flink \(p. 268\)](#) for Apache Flink for more information.

Amazon MSK

You can send data from Amazon MSK to Amazon Timestream by building a data connector similar to the sample Timestream data connector for Kinesis Data Analytics for Apache Flink. Refer to [Amazon Kinesis Data Analytics for Apache flink \(p. 268\)](#) for more information.

Amazon QuickSight

You can use Amazon QuickSight 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 Amazon QuickSight.

Note

All datasets in Amazon QuickSight are read-only. You can't make any changes to your actual data in Timestream by using Amazon QuickSight to remove the data source, dataset, or fields.

Topics

- [Accessing Amazon Timestream from QuickSight \(p. 270\)](#)
- [Create a new QuickSight data source connection for Timestream \(p. 270\)](#)
- [Edit permissions for the QuickSight data source connection for Timestream \(p. 271\)](#)
- [Create a new QuickSight dataset for Timestream \(p. 271\)](#)
- [Create a new analysis for Timestream \(p. 272\)](#)
- [Video tutorial \(p. 272\)](#)

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

1. Ensure you have configured the appropriate permissions for Amazon QuickSight to access Amazon Timestream, as described in [Accessing Amazon Timestream from QuickSight \(p. 270\)](#).
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 \(p. 270\)](#).
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 \(p. 270\)](#).
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 \(p. 270\)](#).
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 Amazon QuickSight works with Timestream.

Amazon SageMaker

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 \(p. 23\)](#) and [Create a table \(p. 23\)](#)
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

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 \(p. 274\)](#)
- [Video tutorial \(p. 274\)](#)

Sample application

1. Create a database and a table in Timestream following the instructions described in [Create a database \(p. 23\)](#) 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 \(p. 33\)](#)
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 [Getting started with Amazon Managed Grafana](#) or complete [Install Grafana](#).
7. If installing Grafana instead of using Amazon Managed Grafana, complete [Install the Timestream plugin for Grafana](#).
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 [Create a Grafana managed alerting rule](#)
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.

Open source Telegraf

You can use the Timestream output plugin for Telegraf to write metrics into Timestream directly from open source Telegraf.

This section provides an explanation of how to install Telegraf with the Timestream output plugin, how to run Telegraf with the Timestream output plugin, and how open source Telegraf works with Timestream.

Topics

- [Installing Telegraf with the Timestream output plugin \(p. 275\)](#)
- [Running Telegraf with the Timestream output plugin \(p. 275\)](#)
- [Mapping Telegraf/InfluxDB metrics to the Timestream model \(p. 276\)](#)

Installing Telegraf with the Timestream output plugin

As of version 1.16, the Timestream 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 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 output plugin

You can follow the instructions below to run Telegraf with the Timestream 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 \(p. 23\)](#), [CLI](#), or [SDKs \(p. 30\)](#).
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 \(p. 23\)](#), [CLI](#), or [SDKs \(p. 30\)](#).
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 model

When writing data from Telegraf to Timestream, 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 output plugin for Telegraf offers multiple options for organizing and storing data in Timestream. 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 \(p. 277\)](#)
- [Storing the data in a single table \(p. 277\)](#)

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 adapter will create two tables, namely, `weather` and `airquality`.
- Each table row will contain a single field only.

The resulting Timestream 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
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 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 table will look like this.

weather

time	location	season	<code><single_table_dimension_name_for_telegraf_measurement_name></code>	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 JDBC connection to connect Timestream to your business intelligence tools and other applications, such as [SQL Workbench](#). The Timestream JDBC driver currently supports SSO with Okta and Microsoft Azure AD.

Topics

- [Configuring the JDBC driver for Timestream \(p. 278\)](#)
- [Connection properties \(p. 279\)](#)
- [JDBC URL examples \(p. 283\)](#)
- [Setting up Timestream JDBC single sign-on authentication with Okta \(p. 284\)](#)
- [Setting up Timestream JDBC single sign-on authentication with Microsoft Azure AD \(p. 285\)](#)

Configuring the JDBC driver for Timestream

Follow the steps below to configure the JDBC driver.

Topics

- [Timestream JDBC driver JARs \(p. 278\)](#)
- [Timestream JDBC driver class and URL format \(p. 279\)](#)
- [Sample application \(p. 279\)](#)

Timestream JDBC driver JARs

You can obtain the Timestream JDBC driver via direct download or by adding the driver as a Maven dependency.

- *As a direct download:* To directly download the Timestream 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 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 JDBC driver class and URL format

The driver class for Timestream 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 with JDBC, we've created a fully functional sample application in GitHub.

1. Create a database with sample data following the instructions described [here \(p. 34\)](#).
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 JDBC driver supports the following options:

Topics

- [Basic authentication options \(p. 280\)](#)
- [Standard client info option \(p. 280\)](#)
- [Driver configuration option \(p. 280\)](#)
- [SDK option \(p. 281\)](#)
- [Endpoint configuration option \(p. 281\)](#)
- [Credential provider options \(p. 281\)](#)
- [SAML-based authentication options for Okta \(p. 282\)](#)
- [SAML-based authentication options for Azure AD \(p. 283\)](#)

Note

If none of the properties are provided, the Timestream 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 authentication (MFA) enabled.	NONE

Standard client info option

The following table describes the Standard Client Info Option.

Option	Description	Default
ApplicationName	The name of the application currently utilizing the connection. ApplicationName is used for debugging purposes and will not be communicated to the Timestream 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 JDBC driver to return metadata for	FALSE

Option	Description	Default
	PreparedStatements, but this will incur an additional cost with Timestream when retrieving the metadata.	
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 request before timing out. Non-positive value disables request timeout.	0
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 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 service.	NONE

Credential provider options

The following table describes the available Credential Provider options.

Option	Description	Default
AwsCredentialsProviderClass	One of <code>PropertiesFileCredentialsProvider</code> or <code>InstanceProfileCredentialsProvider</code> to use for authentication.	NONE
CustomCredentialsFilePath	The path to a properties file containing AWS security credentials <code>accessKey</code> and <code>secretKey</code> . This is only required if <code>AwsCredentialsProviderClass</code> is specified as <code>PropertiesFileCredentialsProvider</code> .	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
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 application. AppId can be found in the <code>entityID</code> field provided in the application metadata. Consider the following example: <code>entityID = http://www.okta.com//IdpAppID</code>	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
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 registered application on Azure AD.	NONE
AADClientSecret	The client secret associated with the registered application 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 \(p. 279\)](#), 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>;Region=us-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=<path  
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>;RoleAR
```

Connect using the Azure AD credentials as the authentication method:

```
jdbc:timestream://  
IdpName=AzureAD;IdpUserName=<name>;IdpPassword=<password>;AADApplicationID=<id>;AADClientSecret=<se
```

Connect with a specific endpoint:

```
jdbc:timestream://Endpoint=abc.us-east-1.amazonaws.com;Region=us-east-1
```

Setting up Timestream JDBC single sign-on authentication with Okta

Timestream supports Timestream JDBC single sign-on authentication with Okta. To use Timestream JDBC single sign-on authentication with Okta, complete each of the sections listed below.

Topics

- [Prerequisites \(p. 284\)](#)
- [AWS account federation in Okta \(p. 285\)](#)
- [Setting up Okta for SAML \(p. 285\)](#)

Prerequisites

Ensure that you have met the following prerequisites before using the Timestream JDBC single sign-on authentication with Okta:

- [Admin permissions in AWS to create the identity provider and the roles \(p. 219\).](#)
- An Okta account (Go to <https://www.okta.com/login/> to create an account).
- [Access to Amazon Timestream \(p. 19\).](#)

Now that you have completed the Prerequisites, you may proceed to [AWS account federation in Okta \(p. 285\)](#).

AWS account federation in Okta

The Timestream 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 \(p. 285\)](#).

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 JDBC single sign-on authentication with Microsoft Azure AD

Timestream supports Timestream JDBC single sign-on authentication with Microsoft Azure AD. To use Timestream JDBC single sign-on authentication with Microsoft Azure AD, complete each of the sections listed below.

Topics

- [Prerequisites \(p. 286\)](#)
- [Setting up Azure AD \(p. 286\)](#)
- [Setting up IAM Identity Provider and roles in AWS \(p. 287\)](#)

Prerequisites

Ensure that you have met the following prerequisites before using the Timestream JDBC single sign-on authentication with Microsoft Azure AD:

- [Admin permissions in AWS to create the identity provider and the roles \(p. 219\).](#)
- An Azure Active Directory account (Go to <https://azure.microsoft.com/en-ca/services/active-directory/> to create an account)
- [Access to Amazon Timestream \(p. 19\).](#)

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** 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.

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. Choose **Microsoft Graph** on the Request API permissions page.

17. Choose **Delegated permissions** and select the **User.Read** permission

18. Choose **Add permissions**

19Choose **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 JDBC single sign-on authentication with Microsoft Azure AD:

Topics

- [Create a SAML Identity Provider \(p. 287\)](#)
- [Create an IAM role \(p. 287\)](#)
- [Create an IAM policy \(p. 287\)](#)
- [Provisioning \(p. 288\)](#)

Create a SAML Identity Provider

To create a SAML Identity Provider for the Timestream 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 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
10. Provide a role description
11. Choose **Create Role** to complete

Create an IAM policy

To create an IAM policy for the Timestream 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

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "iam>ListRoles",  
        "iamListAccountAliases"  
      ],  
      "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 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
12. Choose the IAM Azure AD role and the corresponding Azure Identity Provider created in AWS
13. Choose **Assign**

VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon Timestream by creating an *interface VPC endpoint*. For more information, see [VPC endpoints \(AWS PrivateLink\) \(p. 259\)](#).

Best practices

To fully realize the benefits of the Amazon Timestream, 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. As your data grows over time, you'll notice that Timestream's performance 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 \(p. 289\)](#)
- [Security \(p. 299\)](#)
- [Configuring Amazon Timestream \(p. 299\)](#)
- [Data ingestion \(p. 300\)](#)
- [Queries \(p. 301\)](#)
- [Scheduled queries \(p. 302\)](#)
- [Client applications and supported integrations \(p. 302\)](#)
- [General \(p. 302\)](#)

Data modeling

Amazon Timestream 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 must have temporal characteristics and time must be a quintessential component of the data.

Timestream 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 [Timestream concepts](#) such as dimensions and measures. In this section, you will learn more about:

When deciding whether to create a single table or multiple tables to store data consider the following:

- 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's multi-measure vs. 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 \(p. 290\)](#)
- [Multi-measure records vs. single-measure records \(p. 290\)](#)
- [Dimensions and measures \(p. 292\)](#)
- [Using measure name with multi-measure records \(p. 294\)](#)
- [Recommendations for partitioning multi-measure records \(p. 296\)](#)

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 are abstractions for access control, specifying KMS keys, retention periods, etc. Timestream automatically partition 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 can scale to petabytes of data stored, tens of gigabytes/sec of data writes, and queries can process hundreds of TBs per hour. Queries in Timestream 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. Below are some important considerations when deciding which data to co-locate in the same table vs. 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 need to be in different tables.
- Amazon KMS keys used to encrypt your data is configured at the database level. Therefore, different encryption key requirements imply the data will need to be in different databases.
- Timestream 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 which you frequently query together in the same table, that will generally ease the way you write your queries such 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 continent 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 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 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 table. 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 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 your application is emitting metrics such as `cpu`, `memory`, `disk_iops` from a device measured at the same time instant. Below is an example of such a table where multiple metrics emitted at the same time instant are stored in the same row. You will 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 emit metrics/events at different time periods, for instance when a device's reading/state changes. Since every measure has a unique timestamp, they can be stored in their own records in Timestream. For instance, consider an IoT sensor tracking soil temperature and moisture which emits a record only when it detects a change from the previous reported entry. The example below 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
sensor-sea478	moisture	2021-12-01 19:00:01	NULL	23

Comparing single measure and multi-measure records

Timestream 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% 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 lesser 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 10MB data, both single measure and multi-measure records are comparable. For queries accessing a single measure and scanning > 10MB data, single measure records usually results in lower bytes metered. For queries referencing 3 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 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 (e.g., storing request IDs, additional timestamps, etc, which is discussed below).

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, etc.

Dimensions and measures

A table in Timestream allows you to store *dimensions* (identifying attributes of the device/data you are storing) and *measures* (the metrics/values you are tracking), see [Timestream concepts](#) for more details. As you are modeling your application on Timestream, how you map your data into dimensions and measures impacts your ingestion and query latency. Below are some 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, i.e., attributes that does not change over time. For instance, if you have a server emitting metrics, then the attributes identifying the server, such as host name, region, rack, availability zone, etc are candidate for dimensions. Similarly, for an IoT device with multiple sensors reporting time series data, device id, sensor id, etc. are candidate 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, values, measure name, and timestamp uniquely identifies the time series data. Timestream uses this unique identifier to automatically de-duplicate data. That is, if Timestream receives two data points with the same values of dimension names, dimension values, measure name, and timestamp, if the values have the same version number, then Timestream deduplicates. If the new write request has a lower version than data already existing in Timestream, 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, measure value can be. So any data that might need updates is better modeled as measure values. For instance, if you have a machine in the factory floor whose color can change, you can model the color as a measure value, unless you want to use the color also as identifying attribute that is needed for deduplication. That is, measure values can be used to store attributes that only slowly change over time.

Note that a table in Timestream 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 additional examples below, 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 an unique identifier for every data point (e.g., 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 if 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 (i.e., 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 dimensions 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, it naturally fits different types of attributes:

- *Location information:* For instance, if you want to track the 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 latitude 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 support a special attribute (or column) called measure name. You specify a value for this attribute for every record you write to Timestream. For single measure records, it is natural to use the name of your metric (such as cpu, memory for server metrics, or temperature, pressure, for sensor metrics). When using multi-measure records, since attributes in a multi-measure record are named, and these names become column names in the table, cpu, memory or temperature, pressure can become multi-measure attribute names. So a natural question is how to effectively use the measure name.

Timestream 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 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 can effectively prune data for measure names not relevant to the query, i.e., 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 table. Please be aware of the [limits](#) on the number of unique measure names allowed in a Timestream table. Also note that a measure name is associated with a measure value data type as well, i.e., a single measure name can only be associated with one type of measure value. That type can be one of DOUBLE, BIGINT, BOOLEAN, VARCHAR, and MULTI. Multi-measure records stored with a measure name will have the data type as 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.

Below we discuss a few different examples on 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, pressure. In addition to the actual values, the sensors also report 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, `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

device_id	measure_name	Time	Quality	Value	Unit
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, e.g., temperature, then you can include a measure name predicate in the query. Below 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 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 to be 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 temperature_quality, temperature_value, temperature_unit, pressure_quality, pressure_value, pressure_unit, etc. 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 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 which 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 (i.e., output of SHOW MEASURES query) is:

measure_name	data_type	Dimensions
events	multi	[{"data_type": "varchar", "dimension_name": "available_regions"}, {"data_type": "varchar", "dimension_name": "microservices"}, {"data_type": "varchar", "dimension_name": "instances"}, {"data_type": "varchar", "dimension_name": "processes"}, {"data_type": "varchar", "dimension_name": "jdk_versions"}, {"data_type": "varchar", "dimension_name": "cell_ids"}, {"data_type": "varchar", "dimension_name": "regions"}, {"data_type": "varchar", "dimension_name": "silos"}]
metrics	multi	[{"data_type": "varchar", "dimension_name": "available_regions"}, {"data_type": "varchar", "dimension_name": "microservices"}, {"data_type": "varchar", "dimension_name": "instances"}, {"data_type": "varchar", "dimension_name": "os_versions"}, {"data_type": "varchar", "dimension_name": "cell_ids"}, {"data_type": "varchar", "dimension_name": "regions"}]

measure_name	data_type	Dimensions
		{"data_type": "varchar", "dimension_name": "silo"}, {"data_type": "varchar", "dimension_name": "instance_type"}, {"data_type": "varchar", "dimension_name": "os_version"}]

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 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

We have seen that there is a growing number of workloads in the time series ecosystem that require to ingest and store massive amount of data and at the same time need 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 name 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` 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 to track 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 8192 different measure names, whatever formula is adopted should not generate more than 8192 different values.
- One approach that we have applied with success for string values is to apply a hashing algorithm to the string value and then divide the results by 8192.

```
UserId = foo
measure_name = abs(hash(UserId)%8192)
```

- We also added `abs()` to remove the sign eliminating the possibility for values to range from -8192 to 8192.
- 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 you include a filtering condition in the predicate that uses the newly derived value of the `measure_name`. For example:
 - `select * from table where measure_name = hash(5318645315464564)%8192 and time between X and Y and userId = 5318645315464564`
- This will minimize the total number of partitions scanned to get you data which will translate in faster queries over time.

time	host_name	location	server_type	cpu_usage	available_me	cpu_temp
2022-09-07 21:48:44 .000000000	host-1235	us-east1	5.8xl	55	16.2	78
R2022-09-07 21:48:44 .000000000	host-3587	us-west1	5.8xl	62	18.1	81
2022-09-07 21:48:45.000000000	host-258743	eu-central	5.8xl	88	9.4	91
2022-09-07 21:48:45 .000000000	host-35654	us-east2	5.8xl	29	24	54
R2022-09-07 21:48:45 .000000000	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 .000000000	host-1235	us-east1	6445	5.8xl	55	16.2	78
R2022-09-07 21:48:44 .000000000	host-3587	us-west1	6399	5.8xl	62	18.1	81
2022-09-07 21:48:45 .000000000	host-258743	eu-central	640	5.8xl	88	9.4	91
2022-09-07 21:48:45 .000000000	host-35654	us-east2	2093	5.8xl	29	24	54
R2022-09-07 21:48:45 .000000000	host-254	us-west1	7051	5.8xl	44	32	48

2. *For realtime 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
>>> print(abs(xxhash.xxh64('HOST-ID-1235').intdigest()%8192))
```

Example Go

```
package main

import (
    "bytes"
    "fmt"
    "github.com/pierrec/xxHash/xxHash64"
)

func main() {
    buf := bytes.NewBufferString("HOST-ID-1235")
    x := xxHash64.New(0x0)
    x.Write(buf.Bytes())
    fmt.Printf("%f\n", x.Sum64() % 8192)
}
```

Example Java

```
import net.jpountz.xxhash.XXHash64;
import net.jpountz.xxhash.XXHashFactory;
import static java.lang.Math.abs;
```

```
import java.nio.ByteBuffer;

public class Main {
    public static void main(String[] args) {
        XXHash64 hasher = XXHashFactory.fastestInstance().hash64();

        String host = "HOST-ID-1235";

        ByteBuffer buf = ByteBuffer.wrap(host.getBytes());

        Long result = hasher.hash(buf, 0L);

        Long partition = abs(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, ensure that encryption keys are secured and are not revoked or made inaccessible.
- Monitor API access logs from Amazon CloudTrail. Audit and revoke any anomalous access pattern from unauthorized users.
- Follow additional guidelines described in [Security best practices for Amazon Timestream \(p. 262\)](#)

Configuring Amazon Timestream

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 arrival data. Late arrival 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, and also from resources with intermittent connectivity e.g. an IoT sensor that is online intermittently.
- If you expect late arrival 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 you plan to run on Timestream 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. The magnetic

store is optimized for fast analytical queries that process medium to large volumes of data sent to Timestream.

- Your data retention period should also be influenced by the cost requirements of your system.

For example, consider a scenario where the late arrival 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 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 from that point onwards. Timestream 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.

Data ingestion

- 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 \(p. 309\)](#). Sending data with a timestamp outside these bounds will result in the data being rejected by Timestream 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. Please 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, 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 \(p. 309\)](#).
 - Use common attributes where possible to reduce data transfer and ingestion costs. Refer to [WriteRecords API](#) for more information
- If you encounter partial client-side failures while writing data to Timestream, 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 is designed to auto-scale to the needs of your application. When Timestream 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 at the same (or increased) rate to enable Timestream to auto-scale to the 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.

Queries

Following are suggested best practices for queries with Amazon Timestream.

- Include only the measure and dimension names essential to query. Adding extraneous columns will increase data scans, which impacts the performance of queries.
- Where possible, push the data computation to Timestream 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 \(p. 326\)](#) and [Aggregate functions \(p. 354\)](#).
- 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 \(p. 354\)](#).
- Use a CASE expression to perform complex aggregations instead of selecting from the same table multiple times. See [The CASE statement \(p. 332\)](#).
- 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 \(p. 326\)](#) and [Interval and duration \(p. 349\)](#).
- 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 \(p. 341\)](#).
- 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 at the same rate to enable Amazon Timestream 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, contact AWS 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 which 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 is plotting hourly aggregates for the past few days which 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 to reduce network latencies and data transfer costs. For more information about working with other services, see [Working with other services \(p. 264\)](#). The following are some other helpful links.

- [Best Practices AWS Development with the AWS SDK for Java](#)
- [Best Practices for working with AWS Lambda function](#)
- [Best Practices for Amazon Kinesis Data Analytics](#)
- [Best practices for creating dashboards in Grafana](#)

General

- Ensure that you follow the [The AWS Well-Architected Framework](#) when using Timestream. This whitepaper provides guidance around best practices in operational excellence, security, reliability, performance efficiency, and cost optimization.

Metering and cost optimization

With Amazon Timestream, you pay only for what you use. Timestream 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 Pricing Calculator](#).

This section describes how metering works for writes, storage and queries in Timestream. 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 \(p. 303\)](#)
- [Storage \(p. 305\)](#)
- [Queries \(p. 305\)](#)
- [Cost optimization \(p. 306\)](#)

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 \(p. 303\)](#)
- [Calculating the number of writes \(p. 304\)](#)

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 \(p. 303\)](#), 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 \(p. 304\)](#)
- [Batching time series events in a write \(p. 304\)](#)
- [Batching time series events and using common attributes in a write \(p. 304\)](#)

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, 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 \(p. 303\)](#)), 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 metered based on the number of bytes scanned by each query with a minimum per query as specified on the [pricing page](#). Amazon Timestream's 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:

- 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 arrival data. Late arrival data is incoming data with a timestamp earlier than the current time.
- 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.
- 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.

Troubleshooting

This section contains information on troubleshooting Timestream.

Topics

- [Handling WriteRecords throttles \(p. 307\)](#)
- [Handling rejected records \(p. 307\)](#)
- [Timestream specific error codes \(p. 307\)](#)

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

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.

Timestream specific error codes

This section contains the specific error codes for Timestream.

Timestream write API errors

InternalServerError

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 query API errors

ValidationException

HTTP Status Code: 400

QueryExecutionException

HTTP Status Code: 400

ConflictException

HTTP Status Code: 409

ThrottlingException

HTTP Status Code: 429

InternalServerError

HTTP Status Code: 500

InvalidEndpointException

HTTP Status Code: 421

Quotas

This section describes the current quotas, also referred to as limits, in Amazon Timestream.

Topics

- [Default quotas \(p. 309\)](#)
- [Supported data types \(p. 311\)](#)
- [Naming constraints \(p. 311\)](#)
- [Reserved keywords \(p. 312\)](#)
- [System identifiers \(p. 314\)](#)

Default quotas

The following table contains the Timestream quotas and the default values. To edit data retention for a table from the console, see [Edit a table \(p. 25\)](#).

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
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 will accept data that is up to 15 minutes ahead of the current system time.	15
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

displayName	Description	defaultValue
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
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
Throttle rate for CRUD APIs	The maximum number of Create/Update/List/Describe/Delete database/table/scheduled query API requests allowed per second per account, in the current region.	1
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 chars 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
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

displayName	Description	defaultValue
Scheduled queries per account	The maximum number of schedule queries you can create per Amazon account.	10000
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
Maximum count of active magnetic store partitions	The maximum number of active magnetic store partitions per database. A partition may remain active for up to 6 hours after receiving ingestion.	250

Supported data types

The following table describes the supported data types for measure and dimension values.

Description	Timestream value
Supported data types for measure values.	Big int, double, string, boolean, MULTI, Timestamp
Supported data types for dimension values.	String

Naming constraints

The following table describes naming constraints.

Description	Timestream 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	<ul style="list-style-type: none"> We recommend you do not use System identifiers (p. 314). Can contain a-z A-Z 0-9 _ (underscore) - (dash) . (dot).

Description	Timestream value
	<ul style="list-style-type: none"> • All names must be encoded as UTF-8, and are case sensitive. <p>Note Table and database names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.</p>
Measure Name	<ul style="list-style-type: none"> • Must not contain System identifiers (p. 314) or colon ':'. • Must not start with a reserved prefix (ts_, measure_value). <p>Note Table and database names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.</p>
Dimension Name	<ul style="list-style-type: none"> • Must not contain System identifiers (p. 314), 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". <p>Note Dimension and measure names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.</p>
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.

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 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.

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 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 \(p. 360\)](#) section.

Topics

- [Supported data types \(p. 315\)](#)
- [Built-in time series functionality \(p. 317\)](#)
- [SQL support \(p. 326\)](#)
- [Logical operators \(p. 328\)](#)
- [Comparison operators \(p. 329\)](#)
- [Comparison functions \(p. 330\)](#)
- [Conditional expressions \(p. 331\)](#)
- [Conversion functions \(p. 333\)](#)
- [Mathematical operators \(p. 333\)](#)
- [Mathematical functions \(p. 334\)](#)
- [String operators \(p. 336\)](#)
- [String functions \(p. 336\)](#)
- [Array operators \(p. 338\)](#)
- [Array functions \(p. 339\)](#)
- [Bitwise functions \(p. 340\)](#)
- [Regular expression functions \(p. 341\)](#)
- [Date / time operators \(p. 343\)](#)
- [Date / time functions \(p. 345\)](#)
- [Aggregate functions \(p. 354\)](#)
- [Window functions \(p. 358\)](#)
- [Sample queries \(p. 360\)](#)

Supported data types

Timestream's query language supports the following 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 .
varchar	Variable length character data with a maximum size of 2KB.
array[<i>T</i> ,...]	Contains one or more elements of a specified data type <i>T</i> , where <i>T</i> can be any of the data types supported in Timestream.
row(<i>T</i> ,...)	Contains one or more named fields of data type <i>T</i> . 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 <i>YYYY-MM-DD</i> , where <i>YYYY</i> is the year, <i>MM</i> is the month, and <i>DD</i> is the day, respectively. The supported range is from 1970-01-01 to 2262-04-11. <i>Example:</i> 1971-02-03
time	Represents the time of day in UTC . The time datatype is represented in the form <i>HH.MM.SS.aaaaaaaaaa</i> . Supports nanosecond precision. <i>Example:</i> 17:02:07.496000000
timestamp	Represents an instance in time using nanosecond precision time in UTC, tracking the time since Unix time. <i>YYYY-MM-DD hh:mm:ss.aaaaaaaaaa</i> Query supports timestamps in the range 1677-09-21 00:12:44.000000000 to 2262-04-11 23:47:16.854775807.
interval	Represents an interval of time as a string literal <i>Xt</i> , composed of two parts, <i>X</i> and <i>t</i> . <i>X</i> is a numeric value greater than or equal to 0, and <i>t</i> is a unit of time like second or hour. The unit is not pluralized. The unit of time <i>t</i> is must be one of the following string literals: <ul style="list-style-type: none">• nanosecond• microsecond• millisecond

Data type	Description
	<ul style="list-style-type: none"> • 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)
	<p><i>Examples:</i></p> <div data-bbox="736 766 1480 830" style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">17s</div> <div data-bbox="736 840 1480 903" style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">12second</div> <div data-bbox="736 914 1480 977" style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">21hour</div> <div data-bbox="736 988 1480 1072" style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">2d</div>
<code>timeseries[row(timestamp, T, ...)]</code>	<p>Represents the values of a measure recorded over a time interval as an array composed of <code>row</code> objects. Each <code>row</code> contains a <code>timestamp</code> and one or more measure values of data type <code>T</code>, where <code>T</code> can be any one of <code>bigint</code>, <code>boolean</code>, <code>double</code>, or <code>varchar</code>. Rows are assorted in ascending order by <code>timestamp</code>. The <code>timeseries</code> datatype represents the values of a measure over time.</p>
unknown	<p>Represents null data.</p>

Built-in time series functionality

Timestream 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 \(p. 318\)](#)
- [Time series functions \(p. 320\)](#)

Timeseries views

Timestream supports the following functions for transforming your data to the `timeseries` data type:

Topics

- [CREATE_TIME_SERIES \(p. 318\)](#)
- [UNNEST \(p. 319\)](#)

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.

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-1234567890abcdef1	cpu_utilization	35.0
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	cpu_utilization	38.2
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	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
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-1234567890abcdef1	time: 2019-12-04

region	az	vpc	instance_id	cpu_utilization
				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_value::double: 45.3}]
us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef	[{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 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 \(p. 320\)](#)
- [Derivatives functions \(p. 322\)](#)
- [Integral functions \(p. 323\)](#)
- [Correlation functions \(p. 323\)](#)
- [Filter and reduce functions \(p. 324\)](#)

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 interpolation functions, as well as sample queries.

Usage information

Function	Output data type	Description
<code>interpolate_linear(timeseries array[timestamp])</code>	<code>timeseries</code>	Fills in missing data using linear interpolation .
<code>interpolate_linear(timeseries timestamp)</code>	<code>double</code>	Fills in missing data using linear interpolation .
<code>interpolate_spline_cubic(timeseries, array[timestamp])</code>	<code>timeseries</code>	Fills in missing data using cubic spline interpolation .

Function	Output data type	Description
interpolate_spline_cubic(timeseries, timestamp)	timeseries	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, double 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, double 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 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 <code>derivative_linear(timeseries, interval)</code> , 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 over a specific time period.

Amazon Timestream supports one variant of integral functions. This section provides usage information for the Timestream integral function, as well as sample queries.

Usage information

Function	Output data type	Description
<code>integral_trapezoidal(timeseries, start, end, interval)</code>	<code>double</code>	Approximates the integral over the specified interval for the <code>timeseries</code> provided, using the trapezoidal rule .

Query examples

Example

Calculate the total volume of requests served 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 correlation functions, as well as sample queries.

Usage information

Function	Output data type	Description
<code>correlate_pearson(timeseries1, timeseries2)</code>	<code>double</code>	Calculates Pearson's correlation coefficient for the two timeseries. The timeseries must have the same timestamps.
<code>correlate_spearman(timeseries1, timeseries2)</code>	<code>double</code>	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 filter and reduce functions, as well as sample queries.

Usage information

Function	Output data type	Description
<code>filter(timeseries(T), function(T, Boolean))</code>	<code>timeseries(T)</code>	Constructs a time series from an the input time series, using values for which the passed function returns <code>true</code> .

Function	Output data type	Description
<code>reduce(timeseries(T), initialState S, inputFunction(S, T, S), outputFunction(S, R))</code>	R	Returns a single value, reduced from the time series. The <code>inputFunction</code> will be invoked on each element in <code>timeseries</code> in order. In addition to taking the current element, <code>inputFunction</code> takes the current state (initially <code>initialState</code>) and returns the new state. The <code>outputFunction</code> will be invoked to turn the final state into the result value. The <code>outputFunction</code> 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 supports some common SQL constructs. You can read more below.

Topics

- [SELECT \(p. 326\)](#)
- [Subquery support \(p. 327\)](#)
- [SHOW statements \(p. 327\)](#)
- [DESCRIBE statements \(p. 328\)](#)

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 \(p. 358\)](#).
- `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.

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.

Logical operators

Timestream 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

A	B	A and b	A or b
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 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 supports the following comparison functions.

Topics

- [greatest\(\) \(p. 330\)](#)
- [least\(\) \(p. 330\)](#)
- [ALL\(\), ANY\(\) and SOME\(\) \(p. 330\)](#)

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.
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 supports the following conditional expressions.

Topics

- The CASE statement (p. 332)
- The IF statement (p. 332)
- The COALESCE statement (p. 333)
- The NULLIF statement (p. 333)

- [The TRY statement \(p. 333\)](#)

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	- null	example 3 true	example 4 false

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 supports the following conversion functions.

Topics

- [cast\(\) \(p. 333\)](#)
- [try_cast\(\) \(p. 333\)](#)

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 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 supports the following mathematical operators.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (integer division performs truncation)
%	Modulus (remainder)

Mathematical functions

Timestream 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.
cosine_similarity(x,y)	double	Returns the cosine similarity between the sparse vectors x and y.
degrees(x)	double	Converts angle x in radians to degrees.
e()	double	Returns the constant Euler's number.
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.

Function	Output data type	Description
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).
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]	<p>Returns the signum function of x, that is:</p> <ul style="list-style-type: none"> • 0 if the argument is 0 • 1 if the argument is greater than 0 • -1 if the argument is less than 0. <p>For double arguments, the function additionally returns:</p> <ul style="list-style-type: none"> • 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-radix 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.
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.

Function	Output data type	Description
<code>sin(x)</code>	double	Returns the sine of x.
<code>tan(x)</code>	double	Returns the tangent of x.
<code>tanh(x)</code>	double	Returns the hyperbolic tangent of x.
<code>infinity()</code>	double	Returns the constant representing positive infinity.
<code>is_finite(x)</code>	boolean	Determine if x is finite.
<code>is_infinite(x)</code>	boolean	Determine if x is infinite.
<code>is_nan(x)</code>	boolean	Determine if x is not-a-number.
<code>nan()</code>	double	Returns the constant representing not-a-number.

String operators

Timestream 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
<code>chr(n)</code>	varchar	Returns the Unicode code point n as a varchar.
<code>codepoint(x)</code>	integer	Returns the Unicode code point of the only character of str.
<code>concat(x1, ..., xN)</code>	varchar	Returns the concatenation of x1, x2, ..., xN.
<code>hamming_distance(x1,x2)</code>	bigint	Returns the Hamming distance of x1 and x2, i.e. the number of positions at which the corresponding characters are different. Note that the two varchar inputs must have the same length.
<code>length(x)</code>	bigint	Returns the length of x in characters.
<code>levenshtein_distance(x1, x2)</code>	bigint	Returns the Levenshtein edit distance of x1 and x2, i.e. the minimum number of single-character edits (insertions,

Function	Output data type	Description
		deletions or substitutions) needed to change x1 into x2.
lower(x)	varchar	Converts x to lowercase.
lpad(x1, bigint size, x2)	varchar	Left 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.
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.
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.

Function	Output data type	Description
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.
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 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 supports the following array functions.

Function	Output data type	Description
array_distinct(x)	array	Remove duplicate values from the array x.
array_intersect(x, y)	array	Returns an array of the elements in the intersection of x and y, without duplicates.
array_union(x, y)	array	Returns an array of the elements in the union of x and y, without duplicates.
array_except(x, y)	array	Returns an array of elements in x but not in y, without duplicates.
array_join(x, delimiter, null_replacement)	varchar	Concatenates the elements of the given array using the delimiter and an optional string to replace nulls.
array_max(x)	same as array elements	Returns the maximum value of input array.
array_min(x)	same as array elements	Returns the minimum value of input array.
array_position(x, element)	bigint	Returns the position of the first occurrence of the element in array x (or 0 if not found).
array_remove(x, element)	array	Remove all elements that equal element from array x.
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.
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.
cardinality(x)	bigint	Returns the size of the array x.
concat(array1, array2, ..., arrayN)	array	Concatenates the arrays array1, array2, ..., arrayN.
element_at(array(E), index)	E	Returns element of array at given index. If index < 0,

Function	Output data type	Description
		element_at accesses elements from the last to the first.
repeat(element, count)	array	Repeat element for count times.
reverse(x)	array	Returns an array which has the reversed order of array x.
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.
sequence(start, stop, step)	array(bigint)	Generate a sequence of integers from start to stop, incrementing by step.
sequence(start, stop)	array(timestamp)	Generate a sequence of timestamps from start date to stop date, incrementing by 1 day.
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.
shuffle(x)	array	Generate a random permutation of the given array x.
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.
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.

Bitwise functions

Timestream 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 is a bit signed integer such as 8 or 64. SELECT bit_count(19, 8)

Function	Output data type	Description
		<p>Example result: 3</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT bit_count(19, 2)</pre> </div> <p>Example result: Number must be representable with the bits specified. 19 can not be represented with 2 bits</p>
bitwise_and(bigint, bigint)	bigint (two's complement)	<p>Returns the bitwise AND of the bigint parameters.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT bitwise_and(12, 7)</pre> </div> <p>Example result: 4</p>
bitwise_not(bigint)	bigint (two's complement)	<p>Returns the bitwise NOT of the bigint parameter.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT bitwise_not(12)</pre> </div> <p>Example result: -13</p>
bitwise_or(bigint, bigint)	bigint (two's complement)	<p>Returns the bitwise OR of the bigint parameters.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT bitwise_or(12, 7)</pre> </div> <p>Example result: 15</p>
bitwise_xor(bigint, bigint)	bigint (two's complement)	<p>Returns the bitwise XOR of the bigint parameters.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT bitwise_xor(12, 7)</pre> </div> <p>Example result: 11</p>

Regular expression functions

The regular expression functions in Timestream support the [Java pattern syntax](#). Timestream 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.
regexp_extract_all(string, pattern, group)	array(varchar)	Finds all occurrences of the regular expression pattern in

Function	Output data type	Description
		string and returns the capturing group number group.
regexp_extract(string, pattern)	varchar	Returns the first substring matched by the regular expression pattern in string.
regexp_extract(string, pattern, group)	varchar	Finds the first occurrence of the regular expression pattern in string and returns the capturing group number group.
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 \$.
regexp_replace(string, pattern)	varchar	Removes every instance of the substring matched by the regular expression pattern from string.
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 (\\$).
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).

Function	Output data type	Description
regexp_split(string, pattern)	array(varchar)	Splits string using the regular expression pattern and returns an array. Trailing empty strings are preserved.

Date / time operators

Note

Timestream does not support negative time values. Any operation resulting in negative time results in error.

Timestream supports the following operations on timestamps, dates, and intervals.

Operator	Description
+	Addition
-	Subtraction

Topics

- [Operations \(p. 343\)](#)
- [Addition \(p. 344\)](#)
- [Subtraction \(p. 344\)](#)

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 \(p. 349\)](#).

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 does not support negative time values. Any operation resulting in negative time results in error.

Timestream uses UTC timezone for date and time. Timestream supports the following functions for date and time.

Topics

- [General and conversion \(p. 345\)](#)
- [Interval and duration \(p. 349\)](#)
- [Formatting and parsing \(p. 351\)](#)
- [Extraction \(p. 352\)](#)

General and conversion

Timestream supports the following general and conversion functions for date and time.

Function	Output data type	Description
current_date	date	<p>Returns current date in UTC. No parentheses used.</p> <pre>SELECT current_date</pre> <p>Example result: 2022-07-07</p> <p>Note This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords (p. 312).</p>
current_time	time	<p>Returns current time in UTC. No parentheses used.</p> <pre>SELECT current_time</pre> <p>Example result: 17:41:52.827000000</p> <p>Note This is also a reserved keyword. For a list of reserved</p>

Function	Output data type	Description
		keywords, see Reserved keywords (p. 312) .
current_timestamp or now()	timestamp	<p>Returns current timestamp in UTC.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT current_timestamp</pre> </div> <p>Example result: 2022-07-07 17:42:32.939000000</p> <p>Note This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords (p. 312).</p>
current_timezone()	varchar The value will be 'UTC.'	<p>Timestream uses UTC timezone for date and time.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT current_timezone()</pre> </div> <p>Example result: UTC</p>
date(varchar(x)), date(timestamp)	date	<div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT date(TIMESTAMP '2022-07-07 17:44:43.771000000')</pre> </div> <p>Example result: 2022-07-07</p>
last_day_of_month(timestamp), last_day_of_month(date)	date	<div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT last_day_of_month(TIMESTAMP '2022-07-07 17:44:43.771000000')</pre> </div> <p>Example result: 2022-07-31</p>
from_iso8601_timestamp(string)	timestamp	<p>Parses the ISO 8601 timestamp into internal timestamp format.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>SELECT from_iso8601_timestamp('2022-06-17T08:03:04:05.000000000')</pre> </div> <p>Example result: 2022-06-17 03:04:05.000000000</p>

Function	Output data type	Description
from_iso8601_date(string)	date	<p>Parses the ISO 8601 date string into internal timestamp format for UTC 00:00:00 of the specified date.</p> <pre>SELECT from_iso8601_date('2022-07-17')</pre> <p>Example result: 2022-07-17</p>
to_iso8601(timestamp), to_iso8601(date)	varchar	<p>Returns an ISO 8601 formatted string for the input.</p> <pre>SELECT to_iso8601(from_iso8601_date('2022-06-17'))</pre> <p>Example result: 2022-06-17</p>
from_milliseconds(bigint)	timestamp	<pre>SELECT from_milliseconds(1)</pre> <p>Example result: 1970-01-01 00:00:00.001000000</p>
from_nanoseconds(bigint)	timestamp	<pre>select from_nanoseconds(300000001)</pre> <p>Example result: 1970-01-01 00:00:00.300000001</p>
from_unixtime(double)	timestamp	<p>Returns a timestamp which corresponds to the provided unixtime.</p> <pre>SELECT from_unixtime(1)</pre> <p>Example result: 1970-01-01 00:00:01.000000000</p>
localtime	time	<p>Returns current time in UTC. No parentheses used.</p> <pre>SELECT localtime</pre> <p>Example result: 17:58:22.654000000</p> <p>Note This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords (p. 312).</p>

Function	Output data type	Description
localtimestamp	timestamp	<p>Returns current timestamp in UTC. No parentheses used.</p> <pre>SELECT localtimestamp</pre> <p>Example result: 2022-07-07 17:59:04.368000000</p> <p>Note This is also a reserved keyword. For a list of reserved keywords, see Reserved keywords (p. 312).</p>
to_milliseconds(interval day to second), to_milliseconds(timestamp)	bigint	<pre>SELECT to_milliseconds(INTERVAL '2' DAY + INTERVAL '3' HOUR)</pre> <p>Example result: 183600000</p> <pre>SELECT to_milliseconds(TIMESTAMP '2022-06-17 17:44:43.771000000')</pre> <p>Example result: 1655487883771</p>
to_nanoseconds(interval day to second), to_nanoseconds(timestamp)	bigint	<pre>SELECT to_nanoseconds(INTERVAL '2' DAY + INTERVAL '3' HOUR)</pre> <p>Example result: 1836000000000000</p> <pre>SELECT to_nanoseconds(TIMESTAMP '2022-06-17 17:44:43.771000678')</pre> <p>Example result: 1655487883771000678</p>

Function	Output data type	Description
to_unixtime(timestamp)	double	<p>Returns unixtime for the provided timestamp.</p> <pre>SELECT to_unixtime('2022-06-17 17:44:43.771000000')</pre> <p>Example result: 1.6554878837710001E9</p>
date_trunc(unit, timestamp)	timestamp	<p>Returns the timestamp truncated to unit, where unit is one of [second, minute, hour, day, week, month, quarter, or year].</p> <pre>SELECT date_trunc('minute', TIMESTAMP '2022-06-17 17:44:43.771000000')</pre> <p>Example result: 2022-06-17 17:44:00.000000000</p>

Interval and duration

Timestream 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	<p>Adds a bigint of units, where unit is one of [second, minute, hour, day, week, month, quarter, or year].</p> <pre>SELECT date_add('hour', 9, TIMESTAMP '2022-06-17 00:00:00')</pre> <p>Example result: 2022-06-17 09:00:00.000000000</p>
date_diff(unit, date, date), date_diff(unit, time, time), date_diff(unit, timestamp, timestamp)	bigint	<p>Returns a difference, where unit is one of [second, minute, hour, day, week, month, quarter, or year].</p> <pre>SELECT date_diff('day', DATE '2020-03-01', DATE '2020-03-02')</pre> <p>Example result: 1</p>

Function	Output data type	Description
parse_duration(string)	interval	<p>Parses the input string to return an interval equivalent.</p> <pre>SELECT parse_duration('42.8ms')</pre> <p>Example result: 0 00:00:00.042800000</p> <pre>SELECT typeof(parse_duration('42.8ms'))</pre> <p>Example result: interval day to second</p>
bin(timestamp, interval)	timestamp	<p>Rounds value down to a multiple of the given bin interval.</p> <pre>SELECT bin(current_timestamp, 1day)</pre> <p>Example result: 2022-08-02 00:00:00.000000000</p>
ago(interval)	timestamp	<p>Returns the value corresponding to current_timestamp interval.</p> <pre>SELECT ago(1d)</pre> <p>Example result: 2022-07-06 21:08:53.245000000</p>
interval literals such as 1h, 1d, and 30m	interval	<p>Interval literals are a convenience for parse_duration(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).</p>

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 supports the following formatting and parsing functions for date and time.

Function	Output data type	Description
<code>date_format(timestamp, varchar(x))</code>	<code>varchar</code>	<p>For more information about the format specifiers used by this function, see https://trino.io/docs/current/functions/datetime.html#mysql-date-functions</p> <pre>SELECT date_format('2019-10-20 10:20:20', '%Y-%m-%d %H:%i:%s')</pre> <p>Example result: 2019-10-20 10:20:20</p>
<code>date_parse(varchar(x), varchar(y))</code>	<code>timestamp</code>	<p>For more information about the format specifiers used by this function, see https://trino.io/docs/current/functions/datetime.html#mysql-date-functions</p> <pre>SELECT date_parse('2019-10-20 10:20:20', '%Y-%m-%d %H:%i:%s')</pre> <p>Example result: 2019-10-20 10:20:20.000000000</p>
<code>format_datetime(timestamp, varchar(x))</code>	<code>varchar</code>	<p>For more information about the format string used by this function, see http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html</p> <pre>SELECT format_datetime(parse_datetime('1968-01-12', 'yyyy-MM-dd HH'), 'yyyy-MM-dd HH')</pre> <p>Example result: 1968-01-13 12</p>

Function	Output data type	Description
parse_datetime(varchar(x), varchar(y))	timestamp	<p>For more information about the format string used by this function, see http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html</p> <pre>SELECT parse_datetime('2019-12-29 10:10 PST', 'uuuu-LL-dd HH:mm z')</pre> <p>Example result: 2019-12-29 18:10:00.000000000</p>

Extraction

Timestream 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	<p>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].</p> <pre>SELECT extract(YEAR FROM '2019-10-12 23:10:34.000000000')</pre> <p>Example result: 2019</p>
day(timestamp), day(date), day(interval day to second)	bigint	<pre>SELECT day('2019-10-12 23:10:34.000000000')</pre> <p>Example result: 12</p>
day_of_month(timestamp), day_of_month(date), day_of_month(interval day to second)	bigint	<pre>SELECT day_of_month('2019-10-12 23:10:34.000000000')</pre> <p>Example result: 12</p>
day_of_week(timestamp), day_of_week(date)	bigint	<pre>SELECT day_of_week('2019-10-12 23:10:34.000000000')</pre>

Function	Output data type	Description
		Example result: 6
day_of_year(timestamp), day_of_year(date)	bigint	<pre>SELECT day_of_year('2019-10-12 23:10:34.000000000')</pre>
		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	<pre>SELECT hour('2019-10-12 23:10:34.000000000')</pre>
		Example result: 23
millisecond(timestamp), millisecond(time), millisecond(interval day to second)	bigint	<pre>SELECT millisecond('2019-10-12 23:10:34.000000000')</pre>
		Example result: 0
minute(timestamp), minute(time), minute(interval day to second)	bigint	<pre>SELECT minute('2019-10-12 23:10:34.000000000')</pre>
		Example result: 10
month(timestamp), month(date), month(interval year to month)	bigint	<pre>SELECT month('2019-10-12 23:10:34.000000000')</pre>
		Example result: 10
nanosecond(timestamp), nanosecond(time), nanosecond(interval day to second)	bigint	<pre>SELECT nanosecond(current_timestamp)</pre>
		Example result: 162000000
quarter(timestamp), quarter(date)	bigint	<pre>SELECT quarter('2019-10-12 23:10:34.000000000')</pre>
		Example result: 4
second(timestamp), second(time), second(interval day to second)	bigint	<pre>SELECT second('2019-10-12 23:10:34.000000000')</pre>
		Example result: 34

Function	Output data type	Description
week(timestamp), week(date)	bigint	<pre>SELECT week('2019-10-12 23:10:34.000000000')</pre> <p>Example result: 41</p>
week_of_year(timestamp), week_of_year(date)	bigint	Alias for week
year(timestamp), year(date), year(interval year to month)	bigint	<pre>SELECT year('2019-10-12 23:10:34.000000000')</pre> <p>Example result: 2019</p>
year_of_week(timestamp), year_of_week(date)	bigint	<pre>SELECT year_of_week('2019-10-12 23:10:34.000000000')</pre> <p>Example result: 2019</p>
yow(timestamp), yow(date)	bigint	Alias for year_of_week

Aggregate functions

Timestream 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.
array_agg(x)	array<[same as input]	Returns an array created from the input x elements.
avg(x)	double	Returns the average (arithmetic mean) of all input values.
bool_and(boolean) every(boolean)	boolean	Returns TRUE if every input value is TRUE, otherwise FALSE.
bool_or(boolean)	boolean	Returns TRUE if any input value is TRUE, otherwise FALSE.
count(*) count(x)	bigint	<p>count(*) returns the number of input rows.</p> <p>count(x) returns the number of non-null input values.</p>
count_if(x)	bigint	Returns the number of TRUE input values.
geometric_mean(x)	double	Returns the geometric mean of all input values.

Function	Output data type	Description
max_by(x, y)	[same as x]	Returns the value of x associated with the maximum value of y over all input values.
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.
min_by(x, y)	[same as x]	Returns the value of x associated with the minimum value of y over all input values.
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.
max(x)	[same as input]	Returns the maximum value of all input values.
max(x, n)	array<[same as x]>	Returns n largest values of all input values of x.
min(x)	[same as input]	Returns the minimum value of all input values.
min(x, n)	array<[same as x]>	Returns n smallest values of all input values of x.
sum(x)	[same as input]	Returns the sum of all input values.
bitwise_and_agg(x)	bigint	Returns the bitwise AND of all input values in 2s complement representation.
bitwise_or_agg(x)	bigint	Returns the bitwise OR of all input values in 2s complement representation.
approx_distinct(x)	bigint	Returns the approximate 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 (approximately normal) error distribution over all possible sets. It does not guarantee an upper bound on the error for any specific input set.

Function	Output data type	Description
approx_distinct(x, e)	bigint	Returns the approximate 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 [0.0040625, 0.26000].
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.
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.
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.

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 percentile set. Each element of the array must be between zero and one, and the array must be constant for all input rows.
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. accuracy must be a value greater than zero and less than one, and it must be constant for all input rows.
corr(y, x)	double	Returns correlation coefficient of input values.
covar_pop(y, x)	double	Returns the population covariance of input values.
covar_samp(y, x)	double	Returns the sample covariance of input values.
regr_intercept(y, x)	double	Returns linear regression intercept of input values. y is the dependent value. x is the independent value.
regr_slope(y, x)	double	Returns linear regression slope of input values. y is the dependent value. x is the independent value.
skewness(x)	double	Returns the skewness of all input values.
stddev_pop(x)	double	Returns the population standard deviation of all input values.

Function	Output data type	Description
stddev_samp(x) stddev(x)	double	Returns the sample standard deviation of all input values.
var_pop(x)	double	Returns the population variance of all input values.
var_samp(x) variance(x)	double	Returns the sample variance of all input values.

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 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.
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

Function	Output data type	Description
		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.
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.
last_value(x)	[same as input]	Returns the last value of the window. This function is scoped to the window frame.
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.

Function	Output data type	Description
<code>lead(x[, offset[, default_value]])</code>	[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.
<code>lag(x[, offset[, default_value]])</code>	[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.

Sample queries

This section includes example use cases of Timestream's query language.

Topics

- [Simple queries \(p. 360\)](#)
- [Queries with time series functions \(p. 361\)](#)
- [Queries with aggregate functions \(p. 365\)](#)

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 \(p. 315\)](#). For more information about how measure values are modeled, see [Multi-measure records vs. single-measure records \(p. 290\)](#).

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 \(p. 361\)](#)

Example dataset and queries

You can use Timestream 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	measure_value::bigint
2019-12-04 19:00:00.0000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	35.1	null
2019-12-04 19:00:00.0000000000	us-east-1	us-east-1a	frontend01	memory_utilization	150	null
2019-12-04 19:00:00.0000000000	us-east-1	us-east-1a	frontend01	network_bytes	null	1,500
2019-12-04 19:00:00.0000000000	us-east-1	us-east-1a	frontend01	network_bytes	null	6,700
2019-12-04 19:00:00.0000000000	us-east-1	us-east-1b	frontend02	cpu_utilization	38.5	null
2019-12-04 19:00:00.0000000000	us-east-1	us-east-1b	frontend02	memory_utilization	180	null
2019-12-04 19:00:00.0000000000	us-east-1	us-east-1b	frontend02	network_bytes	null	23,000

Time	region	az	Hostname	measure_name	measure_value	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	network_bytes	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	75.8	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	network_bytes	null	15,000
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	network_bytes	null	836,000
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	55.2	null
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	null	1,245
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	network_bytes	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	85.5	null
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	network_bytes	null	1,245
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	network_bytes	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	72.0	null
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	network_bytes	null	1,400
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	network_bytes	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	75.4	null
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	network_bytes	null	23

Time	region	az	Hostname	measure_name	measure_value	measure_value::bigint
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	network_bytes	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	81.2	null
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	network_bytes	null	1,450
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	network_bytes	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	86.8	null
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	network_bytes	null	300
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	network_bytes	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 \(p. 365\)](#)
- [Example queries \(p. 365\)](#)

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_capa	load_cap	measure_	measure_	measure_	value::varchar
2019-12-01 19:00:00.000000000	123456789	GMC		Astro	Alpha	100	500	fuel_reading	65.2	null
2019-12-01 19:00:00.000000000	123456789	GMC		Astro	Alpha	100	500	load	400.0	null
2019-12-01 19:00:00.000000000	123456789	GMC		Astro	Alpha	100	500	speed	90.2	null
2019-12-01 19:00:00.000000000	123456789	GMC		Astro	Alpha	100	500	location	null	47.6062 N, 122.3321 W
2019-12-01 19:00:00.000000000	123456789	Kenworth	W900		Alpha	150	1000	fuel_reading	10.1	null
2019-12-01 19:00:00.000000000	123456789	Kenworth	W900		Alpha	150	1000	load	950.3	null
2019-12-01 19:00:00.000000000	123456789	Kenworth	W900		Alpha	150	1000	speed	50.8	null
2019-12-01 19:00:00.000000000	123456789	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

This section contains the API Reference documentation for Amazon Timestream.

Timestream has two APIs: Query and Write.

- The **Write API** allows you to perform operations like table creation, resource tagging, and writing of records to Timestream.
- The **Query API** allows you to perform query operations.

Note

Both APIs include the `DescribeEndpoints` action. *For both Query and Write, the `DescribeEndpoints` action are identical.*

You can read more about each API below, along with data types, common errors and parameters.

Note

For error codes specific to Timestream, see [Timestream specific error codes \(p. 307\)](#) For error codes common to all AWS services, see the [AWS Support section](#).

Topics

- [Actions \(p. 368\)](#)
- [Data Types \(p. 447\)](#)
- [Common Errors \(p. 503\)](#)
- [Common Parameters \(p. 505\)](#)

Actions

The following actions are supported by Amazon Timestream Write:

- [CreateDatabase \(p. 370\)](#)
- [CreateTable \(p. 373\)](#)
- [DeleteDatabase \(p. 377\)](#)
- [DeleteTable \(p. 379\)](#)
- [DescribeDatabase \(p. 381\)](#)
- [DescribeEndpoints \(p. 383\)](#)
- [DescribeTable \(p. 385\)](#)
- [ListDatabases \(p. 388\)](#)
- [ListTables \(p. 391\)](#)
- [ListTagsForResource \(p. 394\)](#)
- [TagResource \(p. 396\)](#)
- [UntagResource \(p. 398\)](#)
- [UpdateDatabase \(p. 400\)](#)
- [UpdateTable \(p. 403\)](#)
- [WriteRecords \(p. 406\)](#)

The following actions are supported by Amazon Timestream Query:

- [CancelQuery \(p. 411\)](#)
- [CreateScheduledQuery \(p. 413\)](#)
- [DeleteScheduledQuery \(p. 418\)](#)
- [DescribeEndpoints \(p. 420\)](#)
- [DescribeScheduledQuery \(p. 422\)](#)
- [ExecuteScheduledQuery \(p. 426\)](#)
- [ListScheduledQueries \(p. 428\)](#)
- [ListTagsForResource \(p. 431\)](#)
- [PrepareQuery \(p. 434\)](#)
- [Query \(p. 437\)](#)
- [TagResource \(p. 442\)](#)
- [UntagResource \(p. 444\)](#)
- [UpdateScheduledQuery \(p. 446\)](#)

Amazon Timestream Write

The following actions are supported by Amazon Timestream Write:

- [CreateDatabase \(p. 370\)](#)
- [CreateTable \(p. 373\)](#)
- [DeleteDatabase \(p. 377\)](#)
- [DeleteTable \(p. 379\)](#)
- [DescribeDatabase \(p. 381\)](#)
- [DescribeEndpoints \(p. 383\)](#)
- [DescribeTable \(p. 385\)](#)
- [ListDatabases \(p. 388\)](#)
- [ListTables \(p. 391\)](#)
- [ListTagsForResource \(p. 394\)](#)
- [TagResource \(p. 396\)](#)
- [UntagResource \(p. 398\)](#)
- [UpdateDatabase \(p. 400\)](#)
- [UpdateTable \(p. 403\)](#)
- [WriteRecords \(p. 406\)](#)

CreateDatabase

Service: Amazon Timestream Write

Creates a new Timestream database. If the AWS KMS key is not specified, the database will be encrypted with a Timestream managed AWS KMS key located in your account. For more information, see [AWS managed keys](#). [Service quotas apply](#). For details, see [code sample](#).

Request Syntax

```
{  
  "DatabaseName": "string",  
  "KmsKeyId": "string",  
  "Tags": [  
    {  
      "Key": "string",  
      "Value": "string"  
    }  
  ]  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

DatabaseName (p. 370)

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Pattern: [a-zA-Z0-9_.-]+

Required: Yes

KmsKeyId (p. 370)

The AWS KMS key for the database. If the AWS KMS key is not specified, the database will be encrypted with a Timestream managed AWS KMS key located in your account. For more information, see [AWS managed keys](#).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

Tags (p. 370)

A list of key-value pairs to label the table.

Type: Array of [Tag \(p. 467\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Required: No

Response Syntax

```
{  
  "Database": {  
    "Arn": "string",  
    "CreationTime": number,  
    "DatabaseName": "string",  
    "KmsKeyId": "string",  
    "LastUpdatedTime": number,  
    "TableCount": number  
  }  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Database (p. 371)

The newly created Timestream database.

Type: [Database \(p. 450\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

ConflictException

Timestream was unable to process this request because it contains resource that already exists.

HTTP Status Code: 400

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ServiceQuotaExceededException

The instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

CreateTable

Service: Amazon Timestream Write

Adds a new table to an existing database in your account. In an AWS account, table names must be at least unique within each Region if they are in the same database. You might have identical table names in the same Region if the tables are in separate databases. While creating the table, you must specify the table name, database name, and the retention properties. [Service quotas apply](#). See [code sample](#) for details.

Request Syntax

```
{  
  "DatabaseName": "string",  
  "MagneticStoreWriteProperties": {  
    "EnableMagneticStoreWrites": boolean,  
    "MagneticStoreRejectedDataLocation": {  
      "S3Configuration": {  
        "BucketName": "string",  
        "EncryptionOption": "string",  
        "KmsKeyId": "string",  
        "ObjectKeyPrefix": "string"  
      }  
    }  
  },  
  "RetentionProperties": {  
    "MagneticStoreRetentionPeriodInDays": number,  
    "MemoryStoreRetentionPeriodInHours": number  
  },  
  "TableName": "string",  
  "Tags": [  
    {  
      "Key": "string",  
      "Value": "string"  
    }  
  ]  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[DatabaseName \(p. 373\)](#)

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Pattern: [a-zA-Z0-9_.-]+

Required: Yes

[MagneticStoreWriteProperties \(p. 373\)](#)

Contains properties to set on the table when enabling magnetic store writes.

Type: [MagneticStoreWriteProperties \(p. 455\)](#) object

Required: No

[RetentionProperties \(p. 373\)](#)

The duration for which your time-series data must be stored in the memory store and the magnetic store.

Type: [RetentionProperties \(p. 462\)](#) object

Required: No

[TableName \(p. 373\)](#)

The name of the Timestream table.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Pattern: [a-zA-Z0-9_.-]+

Required: Yes

[Tags \(p. 373\)](#)

A list of key-value pairs to label the table.

Type: Array of [Tag \(p. 467\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Required: No

[Response Syntax](#)

```
{  
  "Table": {  
    "Arn": "string",  
    "CreationTime": number,  
    "DatabaseName": "string",  
    "LastUpdatedTime": number,  
    "MagneticStoreWriteProperties": {  
      "EnableMagneticStoreWrites": boolean,  
      "MagneticStoreRejectedDataLocation": {  
        "S3Configuration": {  
          "BucketName": "string",  
          "EncryptionOption": "string",  
          "KmsKeyId": "string",  
          "ObjectKeyPrefix": "string"  
        }  
      }  
    },  
    "RetentionProperties": {  
      "MagneticStoreRetentionPeriodInDays": number,  
      "MemoryStoreRetentionPeriodInHours": number  
    },  
    "TableName": "string",  
    "TableStatus": "string"  
  }  
}
```

[Response Elements](#)

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Table (p. 374)

The newly created Timestream table.

Type: [Table \(p. 465\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

ConflictException

Timestream was unable to process this request because it contains resource that already exists.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ServiceQuotaExceededException

The instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteDatabase

Service: Amazon Timestream Write

Deletes a given Timestream database. *This is an irreversible operation. After a database is deleted, the time-series data from its tables cannot be recovered.*

Note

All tables in the database must be deleted first, or a ValidationException error will be thrown.

Due to the nature of distributed retries, the operation can return either success or a ResourceNotFoundException. Clients should consider them equivalent.

See [code sample](#) for details.

Request Syntax

```
{  
  "DatabaseName": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

DatabaseName (p. 377)

The name of the Timestream database to be deleted.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteTable

Service: Amazon Timestream Write

Deletes a given Timestream table. This is an irreversible operation. After a Timestream database table is deleted, the time-series data stored in the table cannot be recovered.

Note

Due to the nature of distributed retries, the operation can return either success or a `ResourceNotFoundException`. Clients should consider them equivalent.

See [code sample](#) for details.

Request Syntax

```
{  
  "DatabaseName": "string",  
  "TableName": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

DatabaseName (p. 379)

The name of the database where the Timestream database is to be deleted.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

TableName (p. 379)

The name of the Timestream table to be deleted.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DescribeDatabase

Service: Amazon Timestream Write

Returns information about the database, including the database name, time that the database was created, and the total number of tables found within the database. [Service quotas apply](#). See [code sample](#) for details.

Request Syntax

```
{  
  "DatabaseName": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

DatabaseName (p. 381)

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

Response Syntax

```
{  
  "Database": {  
    "Arn": "string",  
    "CreationTime": number,  
    "DatabaseName": "string",  
    "KmsKeyId": "string",  
    "LastUpdatedTime": number,  
    "TableCount": number  
  }  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Database (p. 381)

The name of the Timestream table.

Type: [Database \(p. 450\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DescribeEndpoints

Service: Amazon Timestream Write

Returns a list of available endpoints to make Timestream API calls against. This API operation is available through both the Write and Query APIs.

Because the Timestream SDKs are designed to transparently work with the service's architecture, including the management and mapping of the service endpoints, *we don't recommend that you use this API operation unless:*

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

For detailed information on how and when to use and implement DescribeEndpoints, see [The Endpoint Discovery Pattern](#).

Response Syntax

```
{  
  "Endpoints": [  
    {  
      "Address": "string",  
      "CachePeriodInMinutes": number  
    }  
  ]  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Endpoints \(p. 383\)](#)

An Endpoints object is returned when a DescribeEndpoints request is made.

Type: Array of [Endpoint \(p. 453\)](#) objects

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DescribeTable

Service: Amazon Timestream Write

Returns information about the table, including the table name, database name, retention duration of the memory store and the magnetic store. [Service quotas apply](#). See [code sample](#) for details.

Request Syntax

```
{  
  "DatabaseName": "string",  
  "TableName": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

DatabaseName (p. 385)

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

TableName (p. 385)

The name of the Timestream table.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

Response Syntax

```
{  
  "Table": {  
    "Arn": "string",  
    "CreationTime": number,  
    "DatabaseName": "string",  
    "LastUpdatedTime": number,  
    "MagneticStoreWriteProperties": {  
      "EnableMagneticStoreWrites": boolean,  
      "MagneticStoreRejectedDataLocation": {  
        "S3Configuration": {  
          "BucketName": "string",  
          "EncryptionOption": "string",  
          "KmsKeyId": "string",  
          "ObjectKeyPrefix": "string"  
        }  
      }  
    }  
  },  
},
```

```
    "RetentionProperties": {  
        "MagneticStoreRetentionPeriodInDays": number,  
        "MemoryStoreRetentionPeriodInHours": number  
    },  
    "TableName": "string",  
    "TableStatus": "string"  
}  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Table (p. 385)

The Timestream table.

Type: [Table \(p. 465\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListDatabases

Service: Amazon Timestream Write

Returns a list of your Timestream databases. [Service quotas apply](#). See [code sample](#) for details.

Request Syntax

```
{  
  "MaxResults": number,  
  "NextToken": string  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[MaxResults \(p. 388\)](#)

The total number of items to return in the output. If the total number of items available is more than the value specified, a NextToken is provided in the output. To resume pagination, provide the NextToken value as argument of a subsequent API invocation.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 20.

Required: No

[NextToken \(p. 388\)](#)

The pagination token. To resume pagination, provide the NextToken value as argument of a subsequent API invocation.

Type: String

Required: No

Response Syntax

```
{  
  "Databases": [  
    {  
      "Arn": string,  
      "CreationTime": number,  
      "DatabaseName": string,  
      "KmsKeyId": string,  
      "LastUpdatedTime": number,  
      "TableCount": number  
    }  
  ],  
  "NextToken": string  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Databases \(p. 388\)](#)

A list of database names.

Type: Array of [Database \(p. 450\)](#) objects

[NextToken \(p. 388\)](#)

The pagination token. This parameter is returned when the response is truncated.

Type: String

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)

- [AWS SDK for Ruby V3](#)

ListTables

Service: Amazon Timestream Write

Provides a list of tables, along with the name, status, and retention properties of each table. See [code sample](#) for details.

Request Syntax

```
{  
  "DatabaseName": "string",  
  "MaxResults": number,  
  "NextToken": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

DatabaseName (p. 391)

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: No

MaxResults (p. 391)

The total number of items to return in the output. If the total number of items available is more than the value specified, a NextToken is provided in the output. To resume pagination, provide the NextToken value as argument of a subsequent API invocation.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 20.

Required: No

NextToken (p. 391)

The pagination token. To resume pagination, provide the NextToken value as argument of a subsequent API invocation.

Type: String

Required: No

Response Syntax

```
{  
  "NextToken": "string",  
  "Tables": [  
    {  
      "Arn": "string",  
      "CreationTime": number,  
      "Name": "string",  
      "Retention": number,  
      "Status": "string"  
    }  
  ]  
}
```

```
  "DatabaseName": "string",
  "LastUpdatedTime": number,
  "MagneticStoreWriteProperties": {
    "EnableMagneticStoreWrites": boolean,
    "MagneticStoreRejectedDataLocation": {
      "S3Configuration": {
        "BucketName": "string",
        "EncryptionOption": "string",
        "KmsKeyId": "string",
        "ObjectKeyPrefix": "string"
      }
    }
  },
  "RetentionProperties": {
    "MagneticStoreRetentionPeriodInDays": number,
    "MemoryStoreRetentionPeriodInHours": number
  },
  "TableName": "string",
  "TableStatus": "string"
}
]
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[NextToken \(p. 391\)](#)

A token to specify where to start paginating. This is the NextToken from a previously truncated response.

Type: String

[Tables \(p. 391\)](#)

A list of tables.

Type: Array of [Table \(p. 465\)](#) objects

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListTagsForResource

Service: Amazon Timestream Write

Lists all tags on a Timestream resource.

Request Syntax

```
{  
  "ResourceARN": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

ResourceARN (p. 394)

The Timestream resource with tags to be listed. This value is an Amazon Resource Name (ARN).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1011.

Required: Yes

Response Syntax

```
{  
  "Tags": [  
    {  
      "Key": "string",  
      "Value": "string"  
    }  
  ]  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Tags (p. 394)

The tags currently associated with the Timestream resource.

Type: Array of [Tag \(p. 467\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

TagResource

Service: Amazon Timestream Write

Associates a set of tags with a Timestream resource. You can then activate these user-defined tags so that they appear on the Billing and Cost Management console for cost allocation tracking.

Request Syntax

```
{  
  "ResourceARN": "string",  
  "Tags": [  
    {  
      "Key": "string",  
      "Value": "string"  
    }  
  ]  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[ResourceARN \(p. 396\)](#)

Identifies the Timestream resource to which tags should be added. This value is an Amazon Resource Name (ARN).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1011.

Required: Yes

[Tags \(p. 396\)](#)

The tags to be assigned to the Timestream resource.

Type: Array of [Tag \(p. 467\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ServiceQuotaExceededException

The instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UntagResource

Service: Amazon Timestream Write

Removes the association of tags from a Timestream resource.

Request Syntax

```
{  
  "ResourceARN": "string",  
  "TagKeys": [ "string" ]  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

ResourceARN (p. 398)

The Timestream resource that the tags will be removed from. This value is an Amazon Resource Name (ARN).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1011.

Required: Yes

TagKeys (p. 398)

A list of tags keys. Existing tags of the resource whose keys are members of this list will be removed from the Timestream resource.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Length Constraints: Minimum length of 1. Maximum length of 128.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ServiceQuotaExceededException

The instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateDatabase

Service: Amazon Timestream Write

Modifies the AWS KMS key for an existing database. While updating the database, you must specify the database name and the identifier of the new AWS KMS key to be used (`KmsKeyId`). If there are any concurrent `UpdateDatabase` requests, first writer wins.

See [code sample](#) for details.

Request Syntax

```
{  
  "DatabaseName": "string",  
  "KmsKeyId": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

DatabaseName (p. 400)

The name of the database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

KmsKeyId (p. 400)

The identifier of the new AWS KMS key (KmsKeyId) to be used to encrypt the data stored in the database. If the KmsKeyId currently registered with the database is the same as the KmsKeyId in the request, there will not be any update.

You can specify the KmsKeyId using any of the following:

- Key ID: 1234abcd-12ab-34cd-56ef-1234567890ab
 - Key ARN: arn:aws:kms:us-east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
 - Alias name: alias/ExampleAlias
 - Alias ARN: arn:aws:kms:us-east-1:111122223333:alias/alias/ExampleAlias

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

Response Syntax

```
    "DatabaseName": "string",
    "KmsKeyId": "string",
    "LastUpdatedTime": number,
    "TableCount": number
}
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Database (p. 400)

A top-level container for a table. Databases and tables are the fundamental management concepts in Amazon Timestream. All tables in a database are encrypted with the same AWS KMS key.

Type: [Database \(p. 450\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ServiceQuotaExceededException

The instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateTable

Service: Amazon Timestream Write

Modifies the retention duration of the memory store and magnetic store for your Timestream table. Note that the change in retention duration takes effect immediately. For example, if the retention period of the memory store was initially set to 2 hours and then changed to 24 hours, 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 does not retrieve data from the magnetic store to populate the memory store.

See [code sample](#) for details.

Request Syntax

```
{  
    "DatabaseName": "string",  
    "MagneticStoreWriteProperties": {  
        "EnableMagneticStoreWrites": boolean,  
        "MagneticStoreRejectedDataLocation": {  
            "S3Configuration": {  
                "BucketName": "string",  
                "EncryptionOption": "string",  
                "KmsKeyId": "string",  
                "ObjectKeyPrefix": "string"  
            }  
        }  
    },  
    "RetentionProperties": {  
        "MagneticStoreRetentionPeriodInDays": number,  
        "MemoryStoreRetentionPeriodInHours": number  
    },  
    "TableName": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[DatabaseName \(p. 403\)](#)

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

[MagneticStoreWriteProperties \(p. 403\)](#)

Contains properties to set on the table when enabling magnetic store writes.

Type: [MagneticStoreWriteProperties \(p. 455\)](#) object

Required: No

[RetentionProperties \(p. 403\)](#)

The retention duration of the memory store and the magnetic store.

Type: [RetentionProperties \(p. 462\)](#) object

Required: No

[TableName \(p. 403\)](#)

The name of the Timestream table.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

Response Syntax

```
{  
  "Table": {  
    "Arn": "string",  
    "CreationTime": number,  
    "DatabaseName": "string",  
    "LastUpdatedTime": number,  
    "MagneticStoreWriteProperties": {  
      "EnableMagneticStoreWrites": boolean,  
      "MagneticStoreRejectedDataLocation": {  
        "S3Configuration": {  
          "BucketName": "string",  
          "EncryptionOption": "string",  
          "KmsKeyId": "string",  
          "ObjectKeyPrefix": "string"  
        }  
      }  
    }  
  },  
  "RetentionProperties": {  
    "MagneticStoreRetentionPeriodInDays": number,  
    "MemoryStoreRetentionPeriodInHours": number  
  },  
  "TableName": "string",  
  "TableStatus": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Table \(p. 404\)](#)

The updated Timestream table.

Type: [Table \(p. 465\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

WriteRecords

Service: Amazon Timestream Write

Enables you to write your time-series data into Timestream. You can specify a single data point or a batch of data points to be inserted into the system. Timestream offers you a flexible schema that auto detects the column names and data types for your Timestream tables based on the dimension names and data types of the data points you specify when invoking writes into the database.

Timestream supports eventual consistency read semantics. This means that when you query data immediately after writing a batch of data into Timestream, the query results might not reflect the results of a recently completed write operation. The results may also include some stale data. If you repeat the query request after a short time, the results should return the latest data. [Service quotas apply](#).

See [code sample](#) for details.

Upserts

You can use the `Version` parameter in a `WriteRecords` request to update data points. Timestream tracks a version number with each record. `Version` defaults to 1 when it's not specified for the record in the request. Timestream updates an existing record's measure value along with its `Version` when it receives a write request with a higher `Version` number for that record. When it receives an update request where the measure value is the same as that of the existing record, Timestream still updates `Version`, if it is greater than the existing value of `Version`. You can update a data point as many times as desired, as long as the value of `Version` continuously increases.

For example, suppose you write a new record without indicating `Version` in the request. Timestream stores this record, and set `Version` to 1. Now, suppose you try to update this record with a `WriteRecords` request of the same record with a different measure value but, like before, do not provide `Version`. In this case, Timestream will reject this update with a `RejectedRecordsException` since the updated record's version is not greater than the existing value of `Version`.

However, if you were to resend the update request with `Version` set to 2, Timestream would then succeed in updating the record's value, and the `Version` would be set to 2. Next, suppose you sent a `WriteRecords` request with this same record and an identical measure value, but with `Version` set to 3. In this case, Timestream would only update `Version` to 3. Any further updates would need to send a version number greater than 3, or the update requests would receive a `RejectedRecordsException`.

Request Syntax

```
{  
  "CommonAttributes": {  
    "Dimensions": [  
      {  
        "DimensionValueType": "string",  
        "Name": "string",  
        "Value": "string"  
      }  
    ],  
    "MeasureName": "string",  
    "MeasureValue": "string",  
    "MeasureValues": [  
      {  
        "Name": "string",  
        "Type": "string",  
        "Value": "string"  
      }  
    ],  
    "MeasureValueType": "string",  
    "Time": "string",  
    "TimeUnit": "string",  
  },  
  "Measure": {  
    "Name": "string",  
    "Type": "string",  
    "Value": "string"  
  },  
  "Time": "string",  
  "TimeUnit": "string",  
  "Version": 1  
}
```

```
        "Version": number
    },
    "DatabaseName": "string",
    "Records": [
        {
            "Dimensions": [
                {
                    "DimensionValueType": "string",
                    "Name": "string",
                    "Value": "string"
                }
            ],
            "MeasureName": "string",
            "MeasureValue": "string",
            "MeasureValues": [
                {
                    "Name": "string",
                    "Type": "string",
                    "Value": "string"
                }
            ],
            "MeasureValueType": "string",
            "Time": "string",
            "TimeUnit": "string",
            "Version": number
        }
    ],
    "TableName": "string"
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[CommonAttributes \(p. 406\)](#)

A record that contains the common measure, dimension, time, and version attributes shared across all the records in the request. The measure and dimension attributes specified will be merged with the measure and dimension attributes in the records object when the data is written into Timestream. Dimensions may not overlap, or a `ValidationException` will be thrown. In other words, a record must contain dimensions with unique names.

Type: [Record \(p. 457\)](#) object

Required: No

[DatabaseName \(p. 406\)](#)

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

[Records \(p. 406\)](#)

An array of records that contain the unique measure, dimension, time, and version attributes for each time-series data point.

Type: Array of [Record \(p. 457\)](#) objects

Array Members: Minimum number of 1 item. Maximum number of 100 items.

Required: Yes

[TableName \(p. 406\)](#)

The name of the Timestream table.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: Yes

Response Syntax

```
{  
  "RecordsIngested": {  
    "MagneticStore": number,  
    "MemoryStore": number,  
    "Total": number  
  }  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[RecordsIngested \(p. 408\)](#)

Information on the records ingested by this request.

Type: [RecordsIngested \(p. 459\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 500

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

RejectedRecordsException

WriteRecords would throw this exception in the following cases:

- Records with duplicate data where 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

In this case, if Timestream rejects data, the `ExistingVersion` field in the `RejectedRecords` response will indicate the current record's version. To force an update, you can resend the request with a version for the record set to a value greater than the `ExistingVersion`.

- Records with timestamps that lie outside the retention duration of the memory store.
- Records with dimensions or measures that exceed the Timestream defined limits.

For more information, see [Quotas](#) in the Amazon Timestream Developer Guide.

HTTP Status Code: 400

ResourceNotFoundException

The operation tried to access a nonexistent resource. The resource might not be specified correctly, or its status might not be ACTIVE.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user and they exceeded the service quotas. The request was throttled.

HTTP Status Code: 400

ValidationException

An invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

Amazon Timestream Query

The following actions are supported by Amazon Timestream Query:

- [CancelQuery \(p. 411\)](#)
- [CreateScheduledQuery \(p. 413\)](#)

- [DeleteScheduledQuery \(p. 418\)](#)
- [DescribeEndpoints \(p. 420\)](#)
- [DescribeScheduledQuery \(p. 422\)](#)
- [ExecuteScheduledQuery \(p. 426\)](#)
- [ListScheduledQueries \(p. 428\)](#)
- [ListTagsForResource \(p. 431\)](#)
- [PrepareQuery \(p. 434\)](#)
- [Query \(p. 437\)](#)
- [TagResource \(p. 442\)](#)
- [UntagResource \(p. 444\)](#)
- [UpdateScheduledQuery \(p. 446\)](#)

CancelQuery

Service: Amazon Timestream Query

Cancels a query that has been issued. Cancellation is provided only if the query has not completed running before the cancellation request was issued. Because cancellation is an idempotent operation, subsequent cancellation requests will return a CancellationMessage, indicating that the query has already been canceled. See [code sample](#) for details.

Request Syntax

```
{  
  "QueryId": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

QueryId (p. 411)

The ID of the query that needs to be cancelled. QueryID is returned as part of the query result.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 64.

Pattern: [a-zA-Z0-9]+

Required: Yes

Response Syntax

```
{  
  "CancellationMessage": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CancellationMessage (p. 411)

A CancellationMessage is returned when a CancelQuery request for the query specified by QueryId has already been issued.

Type: String

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

CreateScheduledQuery

Service: Amazon Timestream Query

Create a scheduled query that will be run on your behalf at the configured schedule. Timestream assumes the execution role provided as part of the `ScheduledQueryExecutionRoleArn` parameter to run the query. You can use the `NotificationConfiguration` parameter to configure notification for your scheduled query operations.

Request Syntax

```
{  
  "ClientToken": "string",  
  "ErrorReportConfiguration": {  
    "S3Configuration": {  
      "BucketName": "string",  
      "EncryptionOption": "string",  
      "ObjectKeyPrefix": "string"  
    }  
  },  
  "KmsKeyId": "string",  
  "Name": "string",  
  "NotificationConfiguration": {  
    "SnsConfiguration": {  
      "TopicArn": "string"  
    }  
  },  
  "QueryString": "string",  
  "ScheduleConfiguration": {  
    "ScheduleExpression": "string"  
  },  
  "ScheduledQueryExecutionRoleArn": "string",  
  "Tags": [  
    {  
      "Key": "string",  
      "Value": "string"  
    }  
  ],  
  "TargetConfiguration": {  
    "TimestreamConfiguration": {  
      "DatabaseName": "string",  
      "DimensionMappings": [  
        {  
          "DimensionValueType": "string",  
          "Name": "string"  
        }  
      ],  
      "MeasureNameColumn": "string",  
      "MixedMeasureMappings": [  
        {  
          "MeasureName": "string",  
          "MeasureValueType": "string",  
          "MultiMeasureAttributeMappings": [  
            {  
              "MeasureValueType": "string",  
              "SourceColumn": "string",  
              "TargetMultiMeasureAttributeName": "string"  
            }  
          ],  
          "SourceColumn": "string",  
          "TargetMeasureName": "string"  
        }  
      ],  
      "MultiMeasureMappings": {  
        "MeasureName": "string",  
        "MeasureValueType": "string",  
        "MultiMeasureAttributeMappings": [  
          {  
            "MeasureValueType": "string",  
            "SourceColumn": "string",  
            "TargetMultiMeasureAttributeName": "string"  
          }  
        ],  
        "SourceColumn": "string",  
        "TargetMeasureName": "string"  
      }  
    }  
  }  
}
```

```
  "MultiMeasureAttributeMappings": [
    {
      "MeasureValueType": "string",
      "SourceColumn": "string",
      "TargetMultiMeasureAttributeName": "string"
    }
  ],
  "TargetMultiMeasureName": "string"
},
"TableName": "string",
"TimeColumn": "string"
}
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[ClientToken \(p. 413\)](#)

Using a ClientToken makes the call to CreateScheduledQuery idempotent, in other words, making the same request repeatedly will produce the same result. Making multiple identical CreateScheduledQuery requests has the same effect as making a single request.

- If CreateScheduledQuery is called without a ClientToken, the Query SDK generates a ClientToken on your behalf.
- After 8 hours, any request with the same ClientToken is treated as a new request.

Type: String

Length Constraints: Minimum length of 32. Maximum length of 128.

Required: No

[ErrorReportConfiguration \(p. 413\)](#)

Configuration for error reporting. Error reports will be generated when a problem is encountered when writing the query results.

Type: [ErrorReportConfiguration \(p. 473\)](#) object

Required: Yes

[KmsKeyId \(p. 413\)](#)

The Amazon KMS key used to encrypt the scheduled query resource, at-rest. If the Amazon KMS key is not specified, the scheduled query resource will be encrypted with a Timestream owned Amazon KMS key. To specify a KMS key, use the key ID, key ARN, alias name, or alias ARN. When using an alias name, prefix the name with *alias/*

If ErrorReportConfiguration uses SSE_KMS as encryption type, the same KmsKeyId is used to encrypt the error report at rest.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

[Name \(p. 413\)](#)

Name of the scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 64.

Pattern: [a-zA-Z0-9_.-]+

Required: Yes

[NotificationConfiguration \(p. 413\)](#)

Notification configuration for the scheduled query. A notification is sent by Timestream when a query run finishes, when the state is updated or when you delete it.

Type: [NotificationConfiguration \(p. 480\)](#) object

Required: Yes

[QueryString \(p. 413\)](#)

The query string to run. Parameter names can be specified in the query string @ character followed by an identifier. The named Parameter @scheduled_runtime is reserved and can be used in the query to get the time at which the query is scheduled to run.

The timestamp calculated according to the ScheduleConfiguration parameter, will be the value of @scheduled_runtime parameter for each query run. For example, consider an instance of a 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.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 262144.

Required: Yes

[ScheduleConfiguration \(p. 413\)](#)

The schedule configuration for the query.

Type: [ScheduleConfiguration \(p. 486\)](#) object

Required: Yes

[ScheduledQueryExecutionRoleArn \(p. 413\)](#)

The ARN for the IAM role that Timestream will assume when running the scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

[Tags \(p. 413\)](#)

A list of key-value pairs to label the scheduled query.

Type: Array of [Tag \(p. 496\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Required: No

[TargetConfiguration \(p. 413\)](#)

Configuration used for writing the result of a query.

Type: [TargetConfiguration \(p. 497\)](#) object

Required: No

Response Syntax

```
{  
  "Arn": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Arn \(p. 416\)](#)

ARN for the created scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

ConflictException

Unable to poll results for a cancelled query.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ServiceQuotaExceededException

You have exceeded the service quota.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteScheduledQuery

Service: Amazon Timestream Query

Deletes a given scheduled query. This is an irreversible operation.

Request Syntax

```
{  
  "ScheduledQueryArn": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

ScheduledQueryArn (p. 418)

The ARN of the scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The requested resource could not be found.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DescribeEndpoints

Service: Amazon Timestream Query

DescribeEndpoints returns a list of available endpoints to make Timestream API calls against. This API is available through both Write and Query.

Because the Timestream SDKs are designed to transparently work with the service's architecture, including the management and mapping of the service endpoints, *it is not recommended that you use this API unless:*

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

For detailed information on how and when to use and implement DescribeEndpoints, see [The Endpoint Discovery Pattern](#).

Response Syntax

```
{  
  "Endpoints": [  
    {  
      "Address": "string",  
      "CachePeriodInMinutes": number  
    }  
  ]  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Endpoints (p. 420)

An Endpoints object is returned when a DescribeEndpoints request is made.

Type: Array of [Endpoint \(p. 472\)](#) objects

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DescribeScheduledQuery

Service: Amazon Timestream Query

Provides detailed information about a scheduled query.

Request Syntax

```
{  
  "ScheduledQueryArn": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

ScheduledQueryArn (p. 422)

The ARN of the scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

Response Syntax

```
{  
  "ScheduledQuery": {  
    "Arn": "string",  
    "CreationTime": number,  
    "ErrorReportConfiguration": {  
      "S3Configuration": {  
        "BucketName": "string",  
        "EncryptionOption": "string",  
        "ObjectKeyPrefix": "string"  
      }  
    },  
    "KmsKeyId": "string",  
    "LastRunSummary": {  
      "ErrorReportLocation": {  
        "S3ReportLocation": {  
          "BucketName": "string",  
          "ObjectKey": "string"  
        }  
      },  
      "ExecutionStats": {  
        "BytesMetered": number,  
        "DataWrites": number,  
        "ExecutionTimeInMillis": number,  
        "QueryResultRows": number,  
        "RecordsIngested": number  
      },  
      "FailureReason": "string",  
      "InvocationTime": number,  
      "RunStatus": "string",  
      "TriggerTime": number  
    }  
  }  
}
```

```
        },
        "Name": "string",
        "NextInvocationTime": number,
        "NotificationConfiguration": {
            "SnsConfiguration": {
                "TopicArn": "string"
            }
        },
        "PreviousInvocationTime": number,
        "QueryString": "string",
        "RecentlyFailedRuns": [
            {
                "ErrorReportLocation": {
                    "S3ReportLocation": {
                        "BucketName": "string",
                        "ObjectKey": "string"
                    }
                },
                "ExecutionStats": {
                    "BytesMetered": number,
                    "DataWrites": number,
                    "ExecutionTimeInMillis": number,
                    "QueryResultRows": number,
                    "RecordsIngested": number
                },
                "FailureReason": "string",
                "InvocationTime": number,
                "RunStatus": "string",
                "TriggerTime": number
            }
        ],
        "ScheduleConfiguration": {
            "ScheduleExpression": "string"
        },
        "ScheduledQueryExecutionRoleArn": "string",
        "State": "string",
        "TargetConfiguration": {
            "TimestreamConfiguration": {
                "DatabaseName": "string",
                "DimensionMappings": [
                    {
                        "DimensionValueType": "string",
                        "Name": "string"
                    }
                ],
                "MeasureNameColumn": "string",
                "MixedMeasureMappings": [
                    {
                        "MeasureName": "string",
                        "MeasureValueType": "string",
                        "MultiMeasureAttributeMappings": [
                            {
                                "MeasureValueType": "string",
                                "SourceColumn": "string",
                                "TargetMultiMeasureAttributeName": "string"
                            }
                        ],
                        "SourceColumn": "string",
                        "TargetMeasureName": "string"
                    }
                ],
                "MultiMeasureMappings": [
                    "MultiMeasureAttributeMappings": [
                        {
                            "MeasureValueType": "string",
                            "SourceColumn": "string",
                            "TargetMultiMeasureAttributeName": "string"
                        }
                    ],
                    "SourceColumn": "string",
                    "TargetMeasureName": "string"
                ]
            }
        }
    ]
}
```

```
        "TargetMultiMeasureAttributeName": "string"
    },
    "TargetMultiMeasureName": "string"
},
"TableName": "string",
"TimeColumnName": "string"
}
}
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[ScheduledQuery \(p. 422\)](#)

The scheduled query.

Type: [ScheduledQueryDescription \(p. 489\)](#) object

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The requested resource could not be found.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ExecuteScheduledQuery

Service: Amazon Timestream Query

You can use this API to run a scheduled query manually.

Request Syntax

```
{  
  "ClientToken": "string",  
  "InvocationTime": number,  
  "ScheduledQueryArn": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[ClientToken \(p. 426\)](#)

Not used.

Type: String

Length Constraints: Minimum length of 32. Maximum length of 128.

Required: No

[InvocationTime \(p. 426\)](#)

The timestamp in UTC. Query will be run as if it was invoked at this timestamp.

Type: Timestamp

Required: Yes

[ScheduledQueryArn \(p. 426\)](#)

ARN of the scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

[AccessDeniedException](#)

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerError

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The requested resource could not be found.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListScheduledQueries

Service: Amazon Timestream Query

Gets a list of all scheduled queries in the caller's Amazon account and Region. `ListScheduledQueries` is eventually consistent.

Request Syntax

```
{  
  "MaxResults": number,  
  "NextToken": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

MaxResults (p. 428)

The maximum number of items to return in the output. If the total number of items available is more than the value specified, a `NextToken` is provided in the output. To resume pagination, provide the `NextToken` value as the argument to the subsequent call to `ListScheduledQueriesRequest`.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 1000.

Required: No

NextToken (p. 428)

A pagination token to resume pagination.

Type: String

Required: No

Response Syntax

```
{  
  "NextToken": "string",  
  "ScheduledQueries": [  
    {  
      "Arn": "string",  
      "CreationTime": number,  
      "ErrorReportConfiguration": {  
        "S3Configuration": {  
          "BucketName": "string",  
          "EncryptionOption": "string",  
          "ObjectKeyPrefix": "string"  
        }  
      },  
      "LastRunStatus": "string",  
      "Name": "string",  
      "NextInvocationTime": number.  
    }  
  ]  
}
```

```
  "PreviousInvocationTime": number,
  "State": "string",
  "TargetDestination": {
    "TimestreamDestination": {
      "DatabaseName": "string",
      "TableName": "string"
    }
  }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[NextToken](#) (p. 428)

A token to specify where to start paginating. This is the NextToken from a previously truncated response.

Type: String

[ScheduledQueries](#) (p. 428)

A list of scheduled queries.

Type: Array of [ScheduledQuery](#) (p. 487) objects

Errors

For information about the errors that are common to all actions, see [Common Errors](#) (p. 503).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListTagsForResource

Service: Amazon Timestream Query

List all tags on a Timestream query resource.

Request Syntax

```
{  
  "MaxResults": number,  
  "NextToken": string,  
  "ResourceARN": string  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[MaxResults \(p. 431\)](#)

The maximum number of tags to return.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 200.

Required: No

[NextToken \(p. 431\)](#)

A pagination token to resume pagination.

Type: String

Required: No

[ResourceARN \(p. 431\)](#)

The Timestream resource with tags to be listed. This value is an Amazon Resource Name (ARN).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

Response Syntax

```
{  
  "NextToken": string,  
  "Tags": [  
    {  
      "Key": string,  
      "Value": string  
    }  
  ]  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

NextToken (p. 431)

A pagination token to resume pagination with a subsequent call to `ListTagsForResourceResponse`.

Type: String

Tags (p. 431)

The tags currently associated with the Timestream resource.

Type: Array of [Tag](#) (p. 496) objects

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Errors

For information about the errors that are common to all actions, see [Common Errors](#) (p. 503).

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The requested resource could not be found.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)

- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PrepareQuery

Service: Amazon Timestream Query

A synchronous operation that allows you to submit a query with parameters to be stored by Timestream for later running. Timestream only supports using this operation with the `PrepareQueryRequest` `$ValidateOnly` set to `true`.

Request Syntax

```
{  
  "QueryString": "string",  
  "ValidateOnly": boolean  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[QueryString \(p. 434\)](#)

The Timestream query string that you want to use as a prepared statement. Parameter names can be specified in the query string @ character followed by an identifier.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 262144.

Required: Yes

[ValidateOnly \(p. 434\)](#)

By setting this value to `true`, Timestream will only validate that the query string is a valid Timestream query, and not store the prepared query for later use.

Type: Boolean

Required: No

Response Syntax

```
{  
  "Columns": [  
    {  
      "Aliased": boolean,  
      "DatabaseName": "string",  
      "Name": "string",  
      "TableName": "string",  
      "Type": {  
        "ColumnInfo": {  
          "Name": "string",  
          "Type": "Type"  
        },  
        "ColumnInfo": [  
          {  
            "Name": "string",  
            "Type": "Type"  
          }  
        ]  
      }  
    }  
  ]  
}
```

```
        ],
        "ScalarType": "string",
        "TimeSeriesMeasureValueColumnInfo": {
            "Name": "string",
            "Type": "Type"
        }
    }
],
"Parameters": [
{
    "Name": "string",
    "Type": {
        "ArrayColumnInfo": {
            "Name": "string",
            "Type": "Type"
        },
        "ColumnInfo": [
            {
                "Name": "string",
                "Type": "Type"
            }
        ],
        "ScalarType": "string",
        "TimeSeriesMeasureValueColumnInfo": {
            "Name": "string",
            "Type": "Type"
        }
    }
}
],
"QueryString": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Columns \(p. 434\)](#)

A list of SELECT clause columns of the submitted query string.

Type: Array of [SelectColumn \(p. 494\)](#) objects

[Parameters \(p. 434\)](#)

A list of parameters used in the submitted query string.

Type: Array of [ParameterMapping \(p. 481\)](#) objects

[QueryString \(p. 434\)](#)

The query string that you want prepare.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 262144.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

Query

Service: Amazon Timestream Query

Query is a synchronous operation that enables you to run a query against your Amazon Timestream data. Query will time out after 60 seconds. You must update the default timeout in the SDK to support a timeout of 60 seconds. See the [code sample](#) for details.

Your query request will fail in the following cases:

- If you submit a Query request with the same client token outside of the 5-minute idempotency window.
- If you submit a Query request with the same client token, but change other parameters, within the 5-minute idempotency window.
- If the size of the row (including the query metadata) exceeds 1 MB, then the query will fail with the following error message:

```
Query aborted as max page response size has been exceeded by the output
result row
```

- If the IAM principal of the query initiator and the result reader are not the same and/or the query initiator and the result reader do not have the same query string in the query requests, the query will fail with an Invalid pagination token error.

Request Syntax

```
{  
  "ClientToken": "string",  
  "MaxRows": number,  
  "NextToken": "string",  
  "QueryString": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[ClientToken \(p. 437\)](#)

Unique, case-sensitive string of up to 64 ASCII characters specified when a Query request is made. Providing a ClientToken makes the call to Query *idempotent*. This means that running the same query repeatedly will produce the same result. In other words, making multiple identical Query requests has the same effect as making a single request. When using ClientToken in a query, note the following:

- If the Query API is instantiated without a ClientToken, the Query SDK generates a ClientToken on your behalf.
- If the Query invocation only contains the ClientToken but does not include a NextToken, that invocation of Query is assumed to be a new query run.
- If the invocation contains NextToken, that particular invocation is assumed to be a subsequent invocation of a prior call to the Query API, and a result set is returned.
- After 4 hours, any request with the same ClientToken is treated as a new request.

Type: String

Length Constraints: Minimum length of 32. Maximum length of 128.

Required: No

[MaxRows \(p. 437\)](#)

The total number of rows to be returned in the Query output. The initial run of Query with a MaxRows value specified will return the result set of the query in two cases:

- The size of the result is less than 1MB.
- The number of rows in the result set is less than the value of maxRows.

Otherwise, the initial invocation of Query only returns a NextToken, which can then be used in subsequent calls to fetch the result set. To resume pagination, provide the NextToken value in the subsequent command.

If the row size is large (e.g. a row has many columns), Timestream may return fewer rows to keep the response size from exceeding the 1 MB limit. If MaxRows is not provided, Timestream will send the necessary number of rows to meet the 1 MB limit.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 1000.

Required: No

[NextToken \(p. 437\)](#)

A pagination token used to return a set of results. When the Query API is invoked using NextToken, that particular invocation is assumed to be a subsequent invocation of a prior call to Query, and a result set is returned. However, if the Query invocation only contains the ClientToken, that invocation of Query is assumed to be a new query run.

Note the following when using NextToken in a query:

- A pagination token can be used for up to five Query invocations, OR for a duration of up to 1 hour – whichever comes first.
- Using the same NextToken will return the same set of records. To keep paginating through the result set, you must use the most recent nextToken.
- Suppose a Query invocation returns two NextToken values, TokenA and TokenB. If TokenB is used in a subsequent Query invocation, then TokenA is invalidated and cannot be reused.
- To request a previous result set from a query after pagination has begun, you must re-invoke the Query API.
- The latest NextToken should be used to paginate until null is returned, at which point a new NextToken should be used.
- If the IAM principal of the query initiator and the result reader are not the same and/or the query initiator and the result reader do not have the same query string in the query requests, the query will fail with an Invalid pagination token error.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

[QueryString \(p. 437\)](#)

The query to be run by Timestream.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 262144.

Required: Yes

Response Syntax

```
{  
  "ColumnInfo": [  
    {  
      "Name": "string",  
      "Type": {  
        "ArrayColumnInfo": "ColumnInfo",  
        "RowColumnInfo": [  
          "ColumnInfo"  
        ],  
        "ScalarType": "string",  
        "TimeSeriesMeasureValueColumnInfo": "ColumnInfo"  
      }  
    }  
  ],  
  "NextToken": "string",  
  "QueryId": "string",  
  "QueryStatus": {  
    "CumulativeBytesMetered": number,  
    "CumulativeBytesScanned": number,  
    "ProgressPercentage": number  
  },  
  "Rows": [  
    {  
      "Data": [  
        {  
          "ArrayValue": [  
            "Datum"  
          ],  
          "NullValue": boolean,  
          "RowValue": "Row",  
          "ScalarValue": "string",  
          "TimeSeriesValue": [  
            {  
              "Time": "string",  
              "Value": "Datum"  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[ColumnInfo \(p. 439\)](#)

The column data types of the returned result set.

Type: Array of [ColumnInfo \(p. 469\)](#) objects

[NextToken \(p. 439\)](#)

A pagination token that can be used again on a Query call to get the next set of results.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

[QueryId \(p. 439\)](#)

A unique ID for the given query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 64.

Pattern: [a-zA-Z0-9]+

[QueryStatus \(p. 439\)](#)

Information about the status of the query, including progress and bytes scanned.

Type: [QueryStatus \(p. 482\)](#) object

[Rows \(p. 439\)](#)

The result set rows returned by the query.

Type: Array of [Row \(p. 483\)](#) objects

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

ConflictException

Unable to poll results for a cancelled query.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

QueryExecutionException

Timestream was unable to run the query successfully.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

TagResource

Service: Amazon Timestream Query

Associate a set of tags with a Timestream resource. You can then activate these user-defined tags so that they appear on the Billing and Cost Management console for cost allocation tracking.

Request Syntax

```
{  
  "ResourceARN": "string",  
  "Tags": [  
    {  
      "Key": "string",  
      "Value": "string"  
    }  
  ]  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

[ResourceARN \(p. 442\)](#)

Identifies the Timestream resource to which tags should be added. This value is an Amazon Resource Name (ARN).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

[Tags \(p. 442\)](#)

The tags to be assigned to the Timestream resource.

Type: Array of [Tag \(p. 496\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The requested resource could not be found.

HTTP Status Code: 400

ServiceQuotaExceededException

You have exceeded the service quota.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UntagResource

Service: Amazon Timestream Query

Removes the association of tags from a Timestream query resource.

Request Syntax

```
{  
  "ResourceARN": "string",  
  "TagKeys": [ "string" ]  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

ResourceARN (p. 444)

The Timestream resource that the tags will be removed from. This value is an Amazon Resource Name (ARN).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

TagKeys (p. 444)

A list of tags keys. Existing tags of the resource whose keys are members of this list will be removed from the Timestream resource.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 200 items.

Length Constraints: Minimum length of 1. Maximum length of 128.

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The requested resource could not be found.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateScheduledQuery

Service: Amazon Timestream Query

Update a scheduled query.

Request Syntax

```
{  
  "ScheduledQueryArn": "string",  
  "State": "string"  
}
```

Request Parameters

For information about the parameters that are common to all actions, see [Common Parameters \(p. 505\)](#).

The request accepts the following data in JSON format.

ScheduledQueryArn (p. 446)

ARN of the scheuled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

State (p. 446)

State of the scheduled query.

Type: String

Valid Values: ENABLED | DISABLED

Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Errors

For information about the errors that are common to all actions, see [Common Errors \(p. 503\)](#).

AccessDeniedException

You are not authorized to perform this action.

HTTP Status Code: 400

InternalServerException

Timestream was unable to fully process this request because of an internal server error.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was not valid.

HTTP Status Code: 400

ResourceNotFoundException

The requested resource could not be found.

HTTP Status Code: 400

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

Invalid or malformed request.

HTTP Status Code: 400

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

Data Types

The following data types are supported by Amazon Timestream Write:

- [Database \(p. 450\)](#)
- [Dimension \(p. 452\)](#)
- [Endpoint \(p. 453\)](#)
- [MagneticStoreRejectedDataLocation \(p. 454\)](#)
- [MagneticStoreWriteProperties \(p. 455\)](#)
- [MeasureValue \(p. 456\)](#)
- [Record \(p. 457\)](#)
- [RecordsIngested \(p. 459\)](#)
- [RejectedRecord \(p. 460\)](#)
- [RetentionProperties \(p. 462\)](#)
- [S3Configuration \(p. 463\)](#)
- [Table \(p. 465\)](#)
- [Tag \(p. 467\)](#)

The following data types are supported by Amazon Timestream Query:

- [ColumnInfo \(p. 469\)](#)
- [Datum \(p. 470\)](#)
- [DimensionMapping \(p. 471\)](#)
- [Endpoint \(p. 472\)](#)
- [ErrorReportConfiguration \(p. 473\)](#)
- [ErrorReportLocation \(p. 474\)](#)
- [ExecutionStats \(p. 475\)](#)
- [MixedMeasureMapping \(p. 476\)](#)
- [MultiMeasureAttributeMapping \(p. 478\)](#)
- [MultiMeasureMappings \(p. 479\)](#)
- [NotificationConfiguration \(p. 480\)](#)
- [ParameterMapping \(p. 481\)](#)
- [QueryStatus \(p. 482\)](#)
- [Row \(p. 483\)](#)
- [S3Configuration \(p. 484\)](#)
- [S3ReportLocation \(p. 485\)](#)
- [ScheduleConfiguration \(p. 486\)](#)
- [ScheduledQuery \(p. 487\)](#)
- [ScheduledQueryDescription \(p. 489\)](#)
- [ScheduledQueryRunSummary \(p. 492\)](#)
- [SelectColumn \(p. 494\)](#)
- [SnsConfiguration \(p. 495\)](#)
- [Tag \(p. 496\)](#)
- [TargetConfiguration \(p. 497\)](#)
- [TargetDestination \(p. 498\)](#)
- [TimeSeriesDataPoint \(p. 499\)](#)
- [TimestreamConfiguration \(p. 500\)](#)
- [TimestreamDestination \(p. 502\)](#)
- [Type \(p. 503\)](#)

Amazon Timestream Write

The following data types are supported by Amazon Timestream Write:

- [Database \(p. 450\)](#)
- [Dimension \(p. 452\)](#)
- [Endpoint \(p. 453\)](#)
- [MagneticStoreRejectedDataLocation \(p. 454\)](#)
- [MagneticStoreWriteProperties \(p. 455\)](#)
- [MeasureValue \(p. 456\)](#)
- [Record \(p. 457\)](#)
- [RecordsIngested \(p. 459\)](#)
- [RejectedRecord \(p. 460\)](#)
- [RetentionProperties \(p. 462\)](#)
- [S3Configuration \(p. 463\)](#)
- [Table \(p. 465\)](#)

- [Tag \(p. 467\)](#)

Database

Service: Amazon Timestream Write

A top-level container for a table. Databases and tables are the fundamental management concepts in Amazon Timestream. All tables in a database are encrypted with the same AWS KMS key.

Contents

Arn

The Amazon Resource Name that uniquely identifies this database.

Type: String

Required: No

CreationTime

The time when the database was created, calculated from the Unix epoch time.

Type: Timestamp

Required: No

DatabaseName

The name of the Timestream database.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: No

KmsKeyId

The identifier of the AWS KMS key used to encrypt the data stored in the database.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

LastUpdatedTime

The last time that this database was updated.

Type: Timestamp

Required: No

TableCount

The total number of tables found within a Timestream database.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Dimension

Service: Amazon Timestream Write

Represents the metadata attributes of the time series. For example, the name and Availability Zone of an EC2 instance or the name of the manufacturer of a wind turbine are dimensions.

Contents

DimensionValueType

The data type of the dimension for the time-series data point.

Type: String

Valid Values: VARCHAR

Required: No

Name

Dimension represents the metadata attributes of the time series. For example, the name and Availability Zone of an EC2 instance or the name of the manufacturer of a wind turbine are dimensions.

For constraints on dimension names, see [Naming Constraints](#).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 60.

Required: Yes

Value

The value of the dimension.

Type: String

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Endpoint

Service: Amazon Timestream Write

Represents an available endpoint against which to make API calls against, as well as the TTL for that endpoint.

Contents

Address

An endpoint address.

Type: String

Required: Yes

CachePeriodInMinutes

The TTL for the endpoint, in minutes.

Type: Long

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

MagneticStoreRejectedDataLocation

Service: Amazon Timestream Write

The location to write error reports for records rejected, asynchronously, during magnetic store writes.

Contents

S3Configuration

Configuration of an S3 location to write error reports for records rejected, asynchronously, during magnetic store writes.

Type: [S3Configuration \(p. 463\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

MagneticStoreWriteProperties

Service: Amazon Timestream Write

The set of properties on a table for configuring magnetic store writes.

Contents

EnableMagneticStoreWrites

A flag to enable magnetic store writes.

Type: Boolean

Required: Yes

MagneticStoreRejectedDataLocation

The location to write error reports for records rejected asynchronously during magnetic store writes.

Type: [MagneticStoreRejectedDataLocation \(p. 454\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

MeasureValue

Service: Amazon Timestream Write

Represents the data attribute of the time series. For example, the CPU utilization of an EC2 instance or the RPM of a wind turbine are measures. MeasureValue has both name and value.

MeasureValue is only allowed for type MULTI. Using MULTI type, you can pass multiple data attributes associated with the same time series in a single record

Contents

Name

The name of the MeasureValue.

For constraints on MeasureValue names, see [Naming Constraints](#) in the Amazon Timestream Developer Guide.

Type: String

Length Constraints: Minimum length of 1.

Required: Yes

Type

Contains the data type of the MeasureValue for the time-series data point.

Type: String

Valid Values: DOUBLE | BIGINT | VARCHAR | BOOLEAN | TIMESTAMP | MULTI

Required: Yes

Value

The value for the MeasureValue.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Record

Service: Amazon Timestream Write

Represents a time-series data point being written into Timestream. Each record contains an array of dimensions. Dimensions represent the metadata attributes of a time-series data point, such as the instance name or Availability Zone of an EC2 instance. A record also contains the measure name, which is the name of the measure being collected (for example, the CPU utilization of an EC2 instance). Additionally, a record contains the measure value and the value type, which is the data type of the measure value. Also, the record contains the timestamp of when the measure was collected and the timestamp unit, which represents the granularity of the timestamp.

Records have a **Version** field, which is a 64-bit long that you can use for updating data points. Writes of a duplicate record with the same dimension, timestamp, and measure name but different measure value will only succeed if the **Version** attribute of the record in the write request is higher than that of the existing record. Timestream defaults to a **Version** of 1 for records without the **Version** field.

Contents

Dimensions

Contains the list of dimensions for time-series data points.

Type: Array of [Dimension \(p. 452\)](#) objects

Array Members: Maximum number of 128 items.

Required: No

MeasureName

Measure represents the data attribute of the time series. For example, the CPU utilization of an EC2 instance or the RPM of a wind turbine are measures.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 256.

Required: No

MeasureValue

Contains the measure value for the time-series data point.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

MeasureValues

Contains the list of MeasureValue for time-series data points.

This is only allowed for type MULTI. For scalar values, use MeasureValue attribute of the record directly.

Type: Array of [MeasureValue \(p. 456\)](#) objects

Required: No

MeasureValueType

Contains the data type of the measure value for the time-series data point. Default type is DOUBLE.

Type: String

Valid Values: DOUBLE | BIGINT | VARCHAR | BOOLEAN | TIMESTAMP | MULTI

Required: No

Time

Contains the time at which the measure value for the data point was collected. The time value plus the unit provides the time elapsed since the epoch. For example, if the time value is 12345 and the unit is ms, then 12345 ms have elapsed since the epoch.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 256.

Required: No

TimeUnit

The granularity of the timestamp unit. It indicates if the time value is in seconds, milliseconds, nanoseconds, or other supported values. Default is MILLISECONDS.

Type: String

Valid Values: MILLISECONDS | SECONDS | MICROSECONDS | NANOSECONDS

Required: No

Version

64-bit attribute used for record updates. Write requests for duplicate data with a higher version number will update the existing measure value and version. In cases where the measure value is the same, Version will still be updated. Default value is 1.

Note

Version must be 1 or greater, or you will receive a ValidationException error.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

RecordsIngested

Service: Amazon Timestream Write

Information on the records ingested by this request.

Contents

MagneticStore

Count of records ingested into the magnetic store.

Type: Integer

Required: No

MemoryStore

Count of records ingested into the memory store.

Type: Integer

Required: No

Total

Total count of successfully ingested records.

Type: Integer

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

RejectedRecord

Service: Amazon Timestream Write

Represents records that were not successfully inserted into Timestream due to data validation issues that must be resolved before reinserting time-series data into the system.

Contents

ExistingVersion

The existing version of the record. This value is populated in scenarios where an identical record exists with a higher version than the version in the write request.

Type: Long

Required: No

Reason

The reason why a record was not successfully inserted into Timestream. Possible causes of failure include:

- Records with duplicate data where 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 Timestream rejects data for this case, the `ExistingVersion` field in the `RejectedRecords` response will indicate the current record's version. To force an update, you can resend the request with a version for the record set to a value greater than the `ExistingVersion`.

- Records with timestamps that lie outside the retention duration of the memory store.

Note

When the retention window is updated, you will receive a `RejectedRecords` exception if you immediately try to ingest data within the new window. To avoid a `RejectedRecords` exception, wait until the duration of the new window to ingest new data. For further information, see [Best Practices for Configuring Timestream](#) and the [explanation of how storage works in Timestream](#).

- Records with dimensions or measures that exceed the Timestream defined limits.

For more information, see [Access Management](#) in the Timestream Developer Guide.

Type: String

Required: No

RecordIndex

The index of the record in the input request for `WriteRecords`. Indexes begin with 0.

Type: Integer

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)

- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

RetentionProperties

Service: Amazon Timestream Write

Retention properties contain the duration for which your time-series data must be stored in the magnetic store and the memory store.

Contents

MagneticStoreRetentionPeriodInDays

The duration for which data must be stored in the magnetic store.

Type: Long

Valid Range: Minimum value of 1. Maximum value of 73000.

Required: Yes

MemoryStoreRetentionPeriodInHours

The duration for which data must be stored in the memory store.

Type: Long

Valid Range: Minimum value of 1. Maximum value of 8766.

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

S3Configuration

Service: Amazon Timestream Write

The configuration that specifies an S3 location.

Contents

BucketName

The bucket name of the customer S3 bucket.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: [a-zA-Z0-9][\.\-\a-zA-Z0-9]{1,61}[a-zA-Z0-9]

Required: No

EncryptionOption

The encryption option for the customer S3 location. Options are S3 server-side encryption with an S3 managed key or AWS managed key.

Type: String

Valid Values: SSE_S3 | SSE_KMS

Required: No

KmsKeyId

The AWS KMS key ID for the customer S3 location when encrypting with an AWS managed key.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

ObjectKeyPrefix

The object key prefix for the customer S3 location.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 928.

Pattern: [a-zA-Z0-9|!\\-_*'\\(\\)]([a-zA-Z0-9]|[_!\\-_*'\\(\\)]\\.)+

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Table

Service: Amazon Timestream Write

Represents a database table in Timestream. Tables contain one or more related time series. You can modify the retention duration of the memory store and the magnetic store for a table.

Contents

Arn

The Amazon Resource Name that uniquely identifies this table.

Type: String

Required: No

CreationTime

The time when the Timestream table was created.

Type: Timestamp

Required: No

DatabaseName

The name of the Timestream database that contains this table.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: No

LastUpdatedTime

The time when the Timestream table was last updated.

Type: Timestamp

Required: No

MagneticStoreWriteProperties

Contains properties to set on the table when enabling magnetic store writes.

Type: [MagneticStoreWriteProperties \(p. 455\)](#) object

Required: No

RetentionProperties

The retention duration for the memory store and magnetic store.

Type: [RetentionProperties \(p. 462\)](#) object

Required: No

TableName

The name of the Timestream table.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 256.

Required: No

TableStatus

The current state of the table:

- **DELETING** - The table is being deleted.
- **ACTIVE** - The table is ready for use.

Type: String

Valid Values: ACTIVE | DELETING

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Tag

Service: Amazon Timestream Write

A tag is a label that you assign to a Timestream database and/or table. Each tag consists of a key and an optional value, both of which you define. With tags, you can categorize databases and/or tables, for example, by purpose, owner, or environment.

Contents

Key

The key of the tag. Tag keys are case sensitive.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Required: Yes

Value

The value of the tag. Tag values are case-sensitive and can be null.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Amazon Timestream Query

The following data types are supported by Amazon Timestream Query:

- [ColumnInfo \(p. 469\)](#)
- [Datum \(p. 470\)](#)
- [DimensionMapping \(p. 471\)](#)
- [Endpoint \(p. 472\)](#)
- [ErrorReportConfiguration \(p. 473\)](#)
- [ErrorReportLocation \(p. 474\)](#)
- [ExecutionStats \(p. 475\)](#)
- [MixedMeasureMapping \(p. 476\)](#)
- [MultiMeasureAttributeMapping \(p. 478\)](#)
- [MultiMeasureMappings \(p. 479\)](#)
- [NotificationConfiguration \(p. 480\)](#)
- [ParameterMapping \(p. 481\)](#)

- [QueryStatus \(p. 482\)](#)
- [Row \(p. 483\)](#)
- [S3Configuration \(p. 484\)](#)
- [S3ReportLocation \(p. 485\)](#)
- [ScheduleConfiguration \(p. 486\)](#)
- [ScheduledQuery \(p. 487\)](#)
- [ScheduledQueryDescription \(p. 489\)](#)
- [ScheduledQueryRunSummary \(p. 492\)](#)
- [SelectColumn \(p. 494\)](#)
- [SnsConfiguration \(p. 495\)](#)
- [Tag \(p. 496\)](#)
- [TargetConfiguration \(p. 497\)](#)
- [TargetDestination \(p. 498\)](#)
- [TimeSeriesDataPoint \(p. 499\)](#)
- [TimestreamConfiguration \(p. 500\)](#)
- [TimestreamDestination \(p. 502\)](#)
- [Type \(p. 503\)](#)

ColumnInfo

Service: Amazon Timestream Query

Contains the metadata for query results such as the column names, data types, and other attributes.

Contents

Name

The name of the result set column. The name of the result set is available for columns of all data types except for arrays.

Type: String

Required: No

Type

The data type of the result set column. The data type can be a scalar or complex. Scalar data types are integers, strings, doubles, Booleans, and others. Complex data types are types such as arrays, rows, and others.

Type: [Type \(p. 503\)](#) object

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Datum

Service: Amazon Timestream Query

Datum represents a single data point in a query result.

Contents

ArrayValue

Indicates if the data point is an array.

Type: Array of [Datum \(p. 470\)](#) objects

Required: No

NullValue

Indicates if the data point is null.

Type: Boolean

Required: No

RowValue

Indicates if the data point is a row.

Type: [Row \(p. 483\)](#) object

Required: No

ScalarValue

Indicates if the data point is a scalar value such as integer, string, double, or Boolean.

Type: String

Required: No

TimeSeriesValue

Indicates if the data point is a timeseries data type.

Type: Array of [TimeSeriesDataPoint \(p. 499\)](#) objects

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

DimensionMapping

Service: Amazon Timestream Query

This type is used to map column(s) from the query result to a dimension in the destination table.

Contents

DimensionValueType

Type for the dimension.

Type: String

Valid Values: VARCHAR

Required: Yes

Name

Column name from query result.

Type: String

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Endpoint

Service: Amazon Timestream Query

Represents an available endpoint against which to make API calls against, as well as the TTL for that endpoint.

Contents

Address

An endpoint address.

Type: String

Required: Yes

CachePeriodInMinutes

The TTL for the endpoint, in minutes.

Type: Long

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ErrorReportConfiguration

Service: Amazon Timestream Query

Configuration required for error reporting.

Contents

S3Configuration

The S3 configuration for the error reports.

Type: [S3Configuration \(p. 484\)](#) object

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ErrorReportLocation

Service: Amazon Timestream Query

This contains the location of the error report for a single scheduled query call.

Contents

S3ReportLocation

The S3 location where error reports are written.

Type: [S3ReportLocation \(p. 485\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ExecutionStats

Service: Amazon Timestream Query

Statistics for a single scheduled query run.

Contents

BytesMetered

Bytes metered for a single scheduled query run.

Type: Long

Required: No

DataWrites

Data writes metered for records ingested in a single scheduled query run.

Type: Long

Required: No

ExecutionTimeInMillis

Total time, measured in milliseconds, that was needed for the scheduled query run to complete.

Type: Long

Required: No

QueryResultRows

Number of rows present in the output from running a query before ingestion to destination data source.

Type: Long

Required: No

RecordsIngested

The number of records ingested for a single scheduled query run.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

MixedMeasureMapping

Service: Amazon Timestream Query

MixedMeasureMappings are mappings that can be used to ingest data into a mixture of narrow and multi measures in the derived table.

Contents

MeasureName

Refers to the value of measure_name in a result row. This field is required if MeasureNameColumn is provided.

Type: String

Required: No

MeasureValueType

Type of the value that is to be read from sourceColumn. If the mapping is for MULTI, use MeasureValueType.MULTI.

Type: String

Valid Values: BIGINT | BOOLEAN | DOUBLE | VARCHAR | MULTI

Required: Yes

MultiMeasureAttributeMappings

Required when measureValueType is MULTI. Attribute mappings for MULTI value measures.

Type: Array of [MultiMeasureAttributeMapping \(p. 478\)](#) objects

Array Members: Minimum number of 1 item.

Required: No

SourceColumn

This field refers to the source column from which measure-value is to be read for result materialization.

Type: String

Required: No

TargetMeasureName

Target measure name to be used. If not provided, the target measure name by default would be measure-name if provided, or sourceColumn otherwise.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)

- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

MultiMeasureAttributeMapping

Service: Amazon Timestream Query

Attribute mapping for MULTI value measures.

Contents

MeasureValueType

Type of the attribute to be read from the source column.

Type: String

Valid Values: BIGINT | BOOLEAN | DOUBLE | VARCHAR | TIMESTAMP

Required: Yes

SourceColumn

Source column from where the attribute value is to be read.

Type: String

Required: Yes

TargetMultiMeasureAttributeName

Custom name to be used for attribute name in derived table. If not provided, source column name would be used.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

MultiMeasureMappings

Service: Amazon Timestream Query

Only one of MixedMeasureMappings or MultiMeasureMappings is to be provided. MultiMeasureMappings can be used to ingest data as multi measures in the derived table.

Contents

MultiMeasureAttributeMappings

Required. Attribute mappings to be used for mapping query results to ingest data for multi-measure attributes.

Type: Array of [MultiMeasureAttributeMapping \(p. 478\)](#) objects

Array Members: Minimum number of 1 item.

Required: Yes

TargetMultiMeasureName

The name of the target multi-measure name in the derived table. This input is required when measureNameColumn is not provided. If MeasureNameColumn is provided, then value from that column will be used as multi-measure name.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

NotificationConfiguration

Service: Amazon Timestream Query

Notification configuration for a scheduled query. A notification is sent by Timestream when a scheduled query is created, its state is updated or when it is deleted.

Contents

SnsConfiguration

Details on SNS configuration.

Type: [SnsConfiguration \(p. 495\)](#) object

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ParameterMapping

Service: Amazon Timestream Query

Mapping for named parameters.

Contents

Name

Parameter name.

Type: String

Required: Yes

Type

Contains the data type of a column in a query result set. The data type can be scalar or complex. The supported scalar data types are integers, Boolean, string, double, timestamp, date, time, and intervals. The supported complex data types are arrays, rows, and timeseries.

Type: [Type \(p. 503\)](#) object

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

QueryStatus

Service: Amazon Timestream Query

Information about the status of the query, including progress and bytes scanned.

Contents

CumulativeBytesMetered

The amount of data scanned by the query in bytes that you will be charged for. This is a cumulative sum and represents the total amount of data that you will be charged for since the query was started. The charge is applied only once and is either applied when the query completes running or when the query is cancelled.

Type: Long

Required: No

CumulativeBytesScanned

The amount of data scanned by the query in bytes. This is a cumulative sum and represents the total amount of bytes scanned since the query was started.

Type: Long

Required: No

ProgressPercentage

The progress of the query, expressed as a percentage.

Type: Double

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Row

Service: Amazon Timestream Query

Represents a single row in the query results.

Contents

Data

List of data points in a single row of the result set.

Type: Array of [Datum \(p. 470\)](#) objects

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

S3Configuration

Service: Amazon Timestream Query

Details on S3 location for error reports that result from running a query.

Contents

BucketName

Name of the S3 bucket under which error reports will be created.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: [a-zA-Z0-9][\.-a-zA-Z0-9]{1,61}[a-zA-Z0-9]

Required: Yes

EncryptionOption

Encryption at rest options for the error reports. If no encryption option is specified, Timestream will choose SSE_S3 as default.

Type: String

Valid Values: SSE_S3 | SSE_KMS

Required: No

ObjectKeyPrefix

Prefix for the error report key. Timestream by default adds the following prefix to the error report path.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 896.

Pattern: [a-zA-Z0-9|!\\-*'\\(\\)]([a-zA-Z0-9]|![\\-*'\\(\\)]/.)+

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

S3ReportLocation

Service: Amazon Timestream Query

S3 report location for the scheduled query run.

Contents

BucketName

S3 bucket name.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: [a-zA-Z0-9][\.-a-zA-Z0-9]{1,61}[a-zA-Z0-9]

Required: No

ObjectKey

S3 key.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ScheduleConfiguration

Service: Amazon Timestream Query

Configuration of the schedule of the query.

Contents

ScheduleExpression

An expression that denotes when to trigger the scheduled query run. This can be a cron expression or a rate expression.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 256.

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ScheduledQuery

Service: Amazon Timestream Query

Scheduled Query

Contents

Arn

The Amazon Resource Name.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

CreationTime

The creation time of the scheduled query.

Type: Timestamp

Required: No

ErrorReportConfiguration

Configuration for scheduled query error reporting.

Type: [ErrorReportConfiguration \(p. 473\)](#) object

Required: No

LastRunStatus

Status of the last scheduled query run.

Type: String

Valid Values: AUTO_TRIGGER_SUCCESS | AUTO_TRIGGER_FAILURE |
MANUAL_TRIGGER_SUCCESS | MANUAL_TRIGGER_FAILURE

Required: No

Name

The name of the scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 64.

Pattern: [a-zA-Z0-9_.-]+

Required: Yes

NextInvocationTime

The next time the scheduled query is to be run.

Type: Timestamp

Required: No

PreviousInvocationTime

The last time the scheduled query was run.

Type: [Timestamp](#)

Required: No

State

State of scheduled query.

Type: [String](#)

Valid Values: [ENABLED](#) | [DISABLED](#)

Required: Yes

TargetDestination

Target data source where final scheduled query result will be written.

Type: [TargetDestination \(p. 498\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ScheduledQueryDescription

Service: Amazon Timestream Query

Structure that describes scheduled query.

Contents

Arn

Scheduled query ARN.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

CreationTime

Creation time of the scheduled query.

Type: Timestamp

Required: No

ErrorReportConfiguration

Error-reporting configuration for the scheduled query.

Type: [ErrorReportConfiguration \(p. 473\)](#) object

Required: No

KmsKeyId

A customer provided KMS key used to encrypt the scheduled query resource.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

LastRunSummary

Runtime summary for the last scheduled query run.

Type: [ScheduledQueryRunSummary \(p. 492\)](#) object

Required: No

Name

Name of the scheduled query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 64.

Pattern: [a-zA-Z0-9_.-]+

Required: Yes

NextInvocationTime

The next time the scheduled query is scheduled to run.

Type: Timestamp

Required: No

NotificationConfiguration

Notification configuration.

Type: [NotificationConfiguration \(p. 480\)](#) object

Required: Yes

PreviousInvocationTime

Last time the query was run.

Type: Timestamp

Required: No

QueryString

The query to be run.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 262144.

Required: Yes

RecentlyFailedRuns

Runtime summary for the last five failed scheduled query runs.

Type: Array of [ScheduledQueryRunSummary \(p. 492\)](#) objects

Required: No

ScheduleConfiguration

Schedule configuration.

Type: [ScheduleConfiguration \(p. 486\)](#) object

Required: Yes

ScheduledQueryExecutionRoleArn

IAM role that Timestream uses to run the schedule query.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

State

State of the scheduled query.

Type: String

Valid Values: ENABLED | DISABLED

Required: Yes

TargetConfiguration

Scheduled query target store configuration.

Type: [TargetConfiguration \(p. 497\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ScheduledQueryRunSummary

Service: Amazon Timestream Query

Run summary for the scheduled query

Contents

ErrorReportLocation

S3 location for error report.

Type: [ErrorReportLocation \(p. 474\)](#) object

Required: No

ExecutionStats

Runtime statistics for a scheduled run.

Type: [ExecutionStats \(p. 475\)](#) object

Required: No

FailureReason

Error message for the scheduled query in case of failure. You might have to look at the error report to get more detailed error reasons.

Type: String

Required: No

InvocationTime

InvocationTime for this run. This is the time at which the query is scheduled to run. Parameter `@scheduled_runtime` can be used in the query to get the value.

Type: Timestamp

Required: No

RunStatus

The status of a scheduled query run.

Type: String

Valid Values: AUTO_TRIGGER_SUCCESS | AUTO_TRIGGER_FAILURE |
MANUAL_TRIGGER_SUCCESS | MANUAL_TRIGGER_FAILURE

Required: No

TriggerTime

The actual time when the query was run.

Type: Timestamp

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

SelectColumn

Service: Amazon Timestream Query

Details of the column that is returned by the query.

Contents

Aliased

True, if the column name was aliased by the query. False otherwise.

Type: Boolean

Required: No

DatabaseName

Database that has this column.

Type: String

Required: No

Name

Name of the column.

Type: String

Required: No

TableName

Table within the database that has this column.

Type: String

Required: No

Type

Contains the data type of a column in a query result set. The data type can be scalar or complex. The supported scalar data types are integers, Boolean, string, double, timestamp, date, time, and intervals. The supported complex data types are arrays, rows, and timeseries.

Type: [Type \(p. 503\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

SnsConfiguration

Service: Amazon Timestream Query

Details on SNS that are required to send the notification.

Contents

TopicArn

SNS topic ARN that the scheduled query status notifications will be sent to.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Tag

Service: Amazon Timestream Query

A tag is a label that you assign to a Timestream database and/or table. Each tag consists of a key and an optional value, both of which you define. Tags enable you to categorize databases and/or tables, for example, by purpose, owner, or environment.

Contents

Key

The key of the tag. Tag keys are case sensitive.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Required: Yes

Value

The value of the tag. Tag values are case sensitive and can be null.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

TargetConfiguration

Service: Amazon Timestream Query

Configuration used for writing the output of a query.

Contents

TimestreamConfiguration

Configuration needed to write data into the Timestream database and table.

Type: [TimestreamConfiguration \(p. 500\)](#) object

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

TargetDestination

Service: Amazon Timestream Query

Destination details to write data for a target data source. Current supported data source is Timestream.

Contents

TimestreamDestination

Query result destination details for Timestream data source.

Type: [TimestreamDestination \(p. 502\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

TimeSeriesDataPoint

Service: Amazon Timestream Query

The timeseries data type represents the values of a measure over time. A time series is an array of rows of timestamps and measure values, with rows sorted in ascending order of time. A TimeSeriesDataPoint is a single data point in the time series. It represents a tuple of (time, measure value) in a time series.

Contents

Time

The timestamp when the measure value was collected.

Type: String

Required: Yes

Value

The measure value for the data point.

Type: [Datum \(p. 470\)](#) object

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

TimestreamConfiguration

Service: Amazon Timestream Query

Configuration to write data into Timestream database and table. This configuration allows the user to map the query result select columns into the destination table columns.

Contents

DatabaseName

Name of Timestream database to which the query result will be written.

Type: String

Required: Yes

DimensionMappings

This is to allow mapping column(s) from the query result to the dimension in the destination table.

Type: Array of [DimensionMapping \(p. 471\)](#) objects

Required: Yes

MeasureNameColumn

Name of the measure column.

Type: String

Required: No

MixedMeasureMappings

Specifies how to map measures to multi-measure records.

Type: Array of [MixedMeasureMapping \(p. 476\)](#) objects

Array Members: Minimum number of 1 item.

Required: No

MultiMeasureMappings

Multi-measure mappings.

Type: [MultiMeasureMappings \(p. 479\)](#) object

Required: No

TableName

Name of Timestream table that the query result will be written to. The table should be within the same database that is provided in Timestream configuration.

Type: String

Required: Yes

TimeColumn

Column from query result that should be used as the time column in destination table. Column type for this should be TIMESTAMP.

Type: String

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

TimestreamDestination

Service: Amazon Timestream Query

Destination for scheduled query.

Contents

DatabaseName

Timestream database name.

Type: String

Required: No

TableName

Timestream table name.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Type

Service: Amazon Timestream Query

Contains the data type of a column in a query result set. The data type can be scalar or complex. The supported scalar data types are integers, Boolean, string, double, timestamp, date, time, and intervals. The supported complex data types are arrays, rows, and timeseries.

Contents

ColumnInfo

Indicates if the column is an array.

Type: [ColumnInfo \(p. 469\)](#) object

Required: No

ColumnInfo

Indicates if the column is a row.

Type: Array of [ColumnInfo \(p. 469\)](#) objects

Required: No

ScalarType

Indicates if the column is of type string, integer, Boolean, double, timestamp, date, time.

Type: String

Valid Values: VARCHAR | BOOLEAN | BIGINT | DOUBLE | TIMESTAMP | DATE | TIME | INTERVAL_DAY_TO_SECOND | INTERVAL_YEAR_TO_MONTH | UNKNOWN | INTEGER

Required: No

TimeSeriesMeasureValueColumnInfo

Indicates if the column is a timeseries data type.

Type: [ColumnInfo \(p. 469\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Common Errors

This section lists the errors common to the API actions of all AWS services. For errors specific to an API action for this service, see the topic for that API action.

AccessDeniedException

You do not have sufficient access to perform this action.

HTTP Status Code: 400

IncompleteSignature

The request signature does not conform to AWS standards.

HTTP Status Code: 400

InternalFailure

The request processing has failed because of an unknown error, exception or failure.

HTTP Status Code: 500

InvalidAction

The action or operation requested is invalid. Verify that the action is typed correctly.

HTTP Status Code: 400

InvalidClientTokenId

The X.509 certificate or AWS access key ID provided does not exist in our records.

HTTP Status Code: 403

InvalidParameterCombination

Parameters that must not be used together were used together.

HTTP Status Code: 400

InvalidParameterValue

An invalid or out-of-range value was supplied for the input parameter.

HTTP Status Code: 400

InvalidQueryParameter

The AWS query string is malformed or does not adhere to AWS standards.

HTTP Status Code: 400

MalformedQueryString

The query string contains a syntax error.

HTTP Status Code: 404

MissingAction

The request is missing an action or a required parameter.

HTTP Status Code: 400

MissingAuthenticationToken

The request must contain either a valid (registered) AWS access key ID or X.509 certificate.

HTTP Status Code: 403

MissingParameter

A required parameter for the specified action is not supplied.

HTTP Status Code: 400

NotAuthorized

You do not have permission to perform this action.

HTTP Status Code: 400

OptInRequired

The AWS access key ID needs a subscription for the service.

HTTP Status Code: 403

RequestExpired

The request reached the service more than 15 minutes after the date stamp on the request or more than 15 minutes after the request expiration date (such as for pre-signed URLs), or the date stamp on the request is more than 15 minutes in the future.

HTTP Status Code: 400

ServiceUnavailable

The request has failed due to a temporary failure of the server.

HTTP Status Code: 503

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

The input fails to satisfy the constraints specified by an AWS service.

HTTP Status Code: 400

Common Parameters

The following list contains the parameters that all actions use for signing Signature Version 4 requests with a query string. Any action-specific parameters are listed in the topic for that action. For more information about Signature Version 4, see [Signature Version 4 Signing Process](#) in the *Amazon Web Services General Reference*.

Action

The action to be performed.

Type: string

Required: Yes

Version

The API version that the request is written for, expressed in the format YYYY-MM-DD.

Type: string

Required: Yes

X-Amz-Algorithm

The hash algorithm that you used to create the request signature.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Valid Values: AWS4-HMAC-SHA256

Required: Conditional

X-Amz-Credential

The credential scope value, which is a string that includes your access key, the date, the region you are targeting, the service you are requesting, and a termination string ("aws4_request"). The value is expressed in the following format: `access_key/YYYYMMDD/region/service/aws4_request`.

For more information, see [Task 2: Create a String to Sign for Signature Version 4](#) in the *Amazon Web Services General Reference*.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

X-Amz-Date

The date that is used to create the signature. The format must be ISO 8601 basic format (YYYYMMDD'T'HHMMSS'Z'). For example, the following date time is a valid X-Amz-Date value: `20120325T120000Z`.

Condition: X-Amz-Date is optional for all requests; it can be used to override the date used for signing requests. If the Date header is specified in the ISO 8601 basic format, X-Amz-Date is not required. When X-Amz-Date is used, it always overrides the value of the Date header. For more information, see [Handling Dates in Signature Version 4](#) in the *Amazon Web Services General Reference*.

Type: string

Required: Conditional

X-Amz-Security-Token

The temporary security token that was obtained through a call to AWS Security Token Service (AWS STS). For a list of services that support temporary security credentials from AWS Security Token Service, go to [AWS Services That Work with IAM](#) in the *IAM User Guide*.

Condition: If you're using temporary security credentials from the AWS Security Token Service, you must include the security token.

Type: string

Required: Conditional

X-Amz-Signature

Specifies the hex-encoded signature that was calculated from the string to sign and the derived signing key.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

X-Amz-SignedHeaders

Specifies all the HTTP headers that were included as part of the canonical request. For more information about specifying signed headers, see [Task 1: Create a Canonical Request For Signature Version 4](#) in the *Amazon Web Services General Reference*.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

Document history

- **Latest documentation update:** April 23, 2021

Change	Description	Date
Amazon Timestream updates to AWS managed policies	New information about AWS managed policies and Amazon Timestream, including updates to existing managed policies.	November 29, 2021
Amazon Timestream supports scheduled queries	Amazon Timestream now supports running a query on your behalf, based on a schedule.	November 29, 2021
Amazon Timestream supports magnetic store.	Amazon Timestream now supports using magnetic storage for your table writes.	November 29, 2021
Amazon Timestream multi-measure records.	Amazon Timestream now supports a more compact format for storing your time-series data.	November 29, 2021
Amazon Timestream updates to AWS managed policies	New information about AWS managed policies and Amazon Timestream, including updates to existing managed policies.	May 24, 2021
Amazon Timestream is now available in the europe (frankfurt) region.	Amazon Timestream is now generally available in the Europe (Frankfurt) region (eu-central-1).	April 23, 2021
Amazon Timestream is now supports VPC endpoints (AWS PrivateLink).	Amazon Timestream now supports the use of VPC endpoints (AWS PrivateLink).	March 23, 2021
Amazon Timestream now supports enhanced query execution statistics.	Amazon Timestream now supports enhanced query execution statistics, such as amount of data scanned.	February 10, 2021
Amazon Timestream now supports cross table queries.	You can use Amazon Timestream to run cross table queries.	February 10, 2021
Amazon Timestream now supports advanced time series functions.	You can use Amazon Timestream to run SQL queries with advanced time series functions, such as derivatives, integrals, and correlations.	February 10, 2021
Amazon Timestream is now HIPAA, ISO, and PCI compliant.	You can now use Amazon Timestream for workloads that require HIPAA, ISO, and PCI-compliant infrastructure.	January 27, 2021

[Amazon Timestream 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.

November 25, 2020

[Amazon Timestream is now generally available.](#)

This documentation covers the initial release of Amazon Timestream.

September 30, 2020