Amazon Timestream

Developer Guide

Amazon Timestream: Developer Guide

Table of Contents

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

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. 24)-** To create a database populated with sample data sets and run sample queries.
- **Timestream Concepts (p. 3)-**To learn essential Timestream concepts.
- **Accessing Timestream (p. 12)-**To learn how to access Timestream using the console, AWS CLI, or API.
- **Quotas (p. 211)-**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. 20)
- Query Language Reference (p. 159)

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. 12) sections, to learn how to access Timestream using the Console, CLI or SDKs.

Topics

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:

- 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.

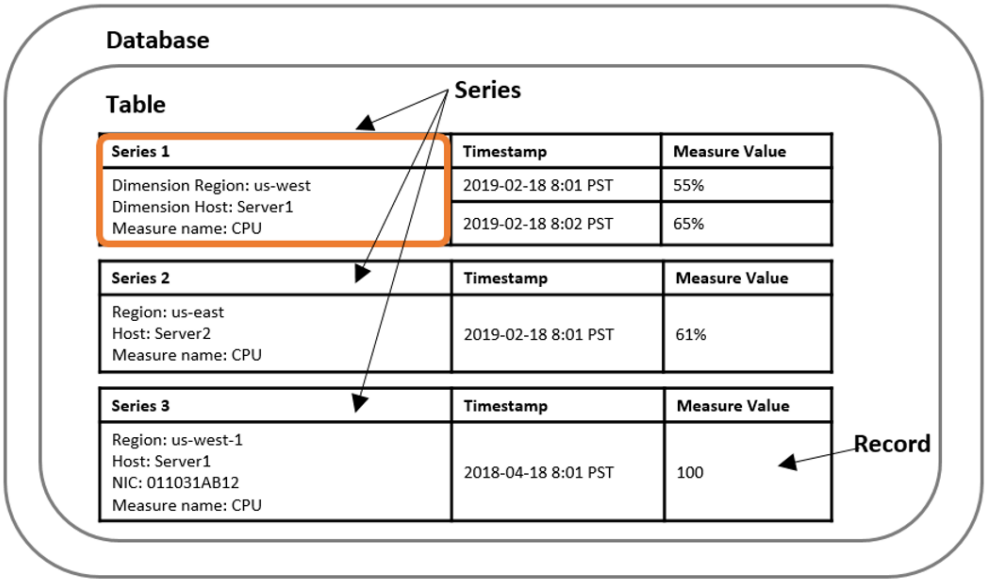
Following are the key concepts of Timestream:

- **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

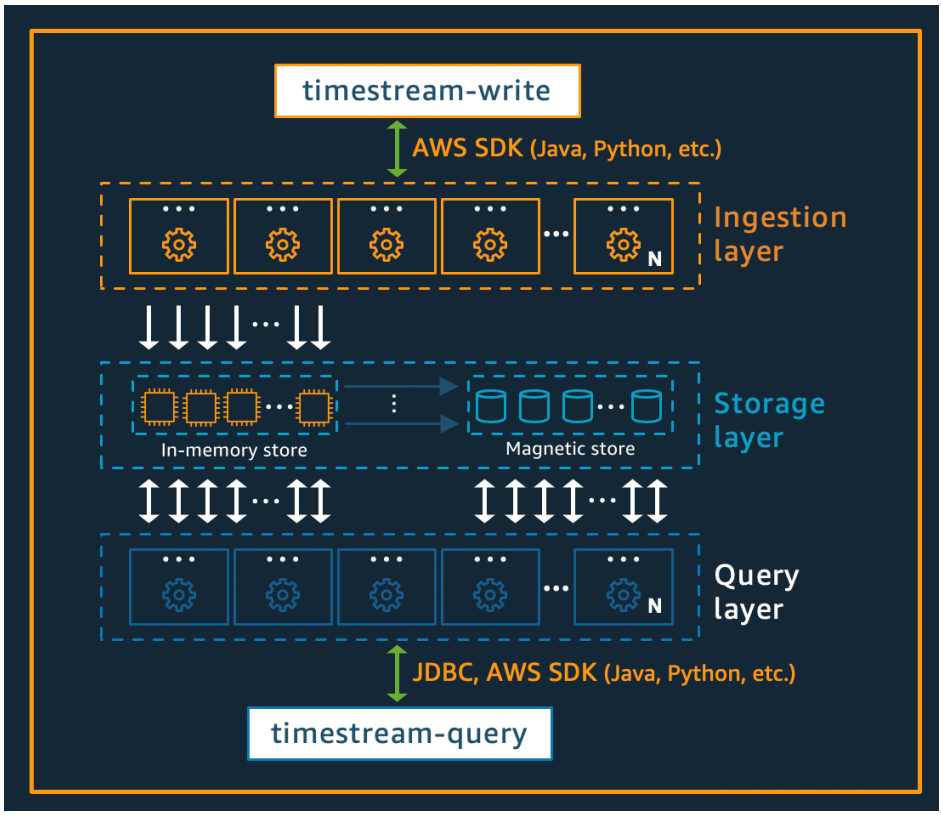
A **database** contains 0 or more **tables**. Each **table** contains 0 or more **time series**. Each **time series** consists of a sequence of **records** over a given time interval at a specified **granularity**. Each **time series** can be described using its meta-data or **dimensions**, its data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

When time-series data is sent to Timestream using the SDKs or via direct invocation of the Write APIs, it is first handled by a data ingestion layer, before being written to a fault tolerant memory store. The data ingestion layer is designed to process trillions of events per second and scale horizontally to match the requirements of your application. The memory store processes incoming data from the ingestion layer, detects duplicates, and replicates the data across three availability zones.

Storage Architecture

When data is stored in Timestream, it is automatically indexed based on its temporal and contextual attributes to ensure optimal data retrieval when queried. Timestream also automates data lifecycle management by enabling you to configure table-specific data retention policies. For example, you can configure a data retention policy to automatically move data from memory store to magnetic store as it reaches a certain age.

Once the data moves to the magnetic store, it is reorganized into a format that is optimized for large volume data reads. Like the memory store, the magnetic store also allows for configuration of data retention policies. For example, you can configure a data retention policy to delete magnetic store data as it reaches a certain age threshold, or store data in the magnetic store for up to 200 years.

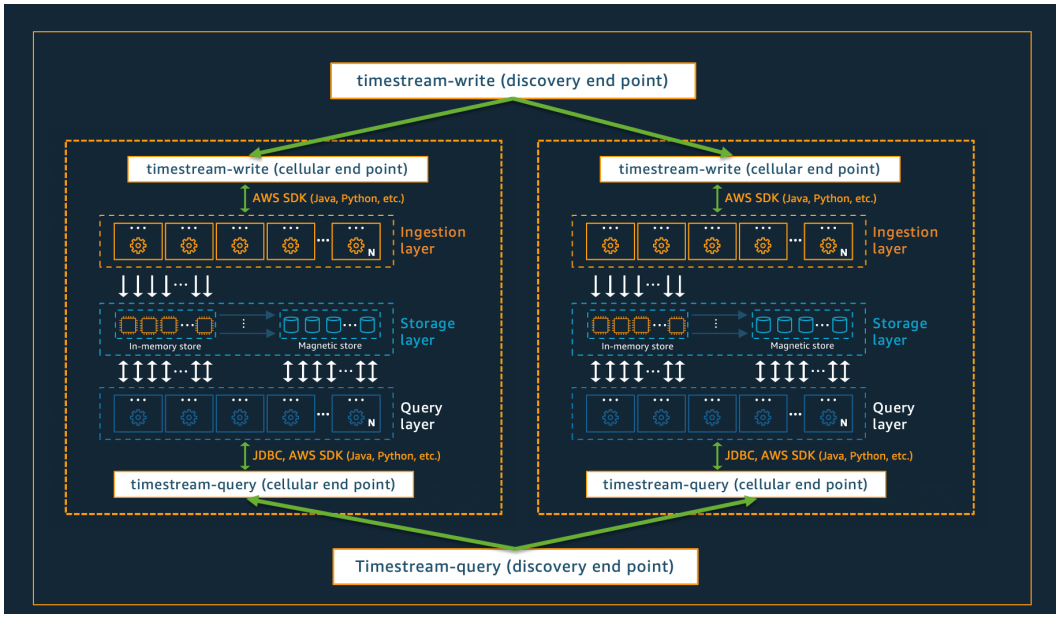
Query Architecture

Timestream simplifies data access through its purpose-built adaptive query engine. The query engine can seamlessly access and combine data across storage tiers without having to specify the location of the data. This allows you to quickly and easily derive insights from your data using SQL queries.

Timestream queries are run by a dedicated fleet of workers (independent of the ingestion and storage nodes), where the number of workers enlisted to run a given query is determined by query complexity and data size. Through massive parallelism on both the query execution fleet and the storage fleets of the system, complex queries over large datasets are highly performant.

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. 128) for instructions on how to set up VPC Endpoints, and Endpoint Discovery Pattern (p. 18) 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	Represent a 64-bit 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.

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 theses query requests after a short time, the results should return the latest data.

You can collect time series data from connected devices, IT systems, industrial equipment, and many other systems, and write it into Amazon Timestream. You can write data using the AWS SDKs (p. 20), AWS CLI (p. 17), or through AWS Lambda (p. 133), AWS IoT Core (p. 135), Amazon Kinesis Data Analytics for Apache Flink (p. 138), Amazon Kinesis (p. 139), Amazon MSK (p. 139) and Open source Telegraf (p. 144).

Topics

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. 16) and Create a table (p. 16). 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.

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. 205) 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. When data is first written to Timestream, it arrives in the memory store. The memory store detects duplicate data values, sorts data, and durably stores the data. The memory store also enables you to run fast point in time queries. Memory store is typically used to store the most recent data. On the other hand, the magnetic store is designed for storing historical data. It enables you to run analytic queries based on the volume of data being queried.

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 CMK keys to secure your data in the magnetic store. For more information on customer managed CMKs, see Customer master Keys.

Query

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. 159) 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. 9)
- Time series model (p. 11)

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. 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-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	35	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	38.2	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	45.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	54.9	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	42.6	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	33.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	54.9	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	42.6	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	33.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	54.9	null
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	42.6	null
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	memory_utilization	33.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes_in	null	30,000
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes_in	null	15,200
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes_in	null	34,400
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes_in	null	1,500
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes_in	null	6,600
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	network_bytes_in	null	7,200

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. 159) for more details.

Using the earlier example of the EC2 instance, here is the same data expressed as a timeseries:

region	az	vpc	instance_id	cpu_utilization
us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	[{time: 2019-12-04 19:00:00.000000000, value: 35}, {time: 2019-12-04 19:00:01.000000000, value: 38.2}, {time: 2019-12-04 19:00:02.000000000, value: 45.3}]

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. 12)
2. Create an IAM user with Timestream access. For more information, see Create an IAM User with Timestream access (p. 12).
3. Get an AWS access key (p. 14) (not required for Console access).
4. Configure your credentials (p. 14) (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.

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 for yourself and add the user to an administrators group (console)

1. Sign in to the IAM console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   Note
   We strongly recommend that you adhere to the best practice of using the **`Administrator`** IAM user that follows and securely lock away the root user credentials. Sign in as the root user only to perform a few account and service management tasks.
2. In the navigation pane, choose **Users** and then choose **Add user**.

3. For **User name**, enter **`Administrator`**.
4. Select the check box next to **AWS Management Console access**. Then select **Custom password**, and then enter your new password in the text box.
5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to **User must create a new password at next sign-in** to allow the new user to reset their password after they sign in.
6. Choose **Next: Permissions**.
7. Under **Set permissions**, choose **Add user to group**.
8. Choose **Create group**.
9. In the **Create group** dialog box, for **Group name** enter **`Administrators`**.
10. Choose **Filter policies**, and then select **AWS managed - job function** to filter the table contents.
11. In the policy list, select the check box for **AdministratorAccess**. Then choose **Create group**.

    Note
    You must activate IAM user and role access to Billing before you can use the `AdministratorAccess` permissions to access the AWS Billing and Cost Management console. To do this, follow the instructions in step 1 of the tutorial about delegating access to the billing console.
12. Back in the list of groups, select the check box for your new group. Choose **Refresh** if necessary to see the group in the list.
13. Choose **Next: Tags**.
14. (Optional) Add metadata to the user by attaching tags as key-value pairs. For more information about using tags in IAM, see Tagging IAM entities in the *IAM User Guide*.
15. Choose **Next: Review** to see the list of group memberships to be added to the new user. When you are ready to proceed, choose **Create user**.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see Access management and Example policies.

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"
            ],
            "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. 17).

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

Using the Console

You can use the AWS Management Console for Timestream to create tables and databases, run queries and delete databases and tables.

Topics

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. 16).
   - **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. 16).
4. In the left pane, select the database created in Create a table (p. 16).
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
```

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. 16).
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 table that you created in **Create a database**.
4. Click **Delete**
5. Type *delete* in the confirmation box.

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. 14).

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

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
```

Using the API

In addition to the SDKs (p. 20), Amazon Timestream provides direct REST API access via the *endpoint discovery pattern*. The endpoint discovery pattern is described below, along with it's 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. 128)
- 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

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. 19).

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. 20).

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
      ```

      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. 27).

Topics

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. 27).

Prerequisites

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

1. Follow the AWS setup instructions in Accessing Timestream (p. 12).
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. 27).

Prerequisites

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

1. Follow the AWS setup instructions in Accessing Timestream (p. 12).
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. 27).

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. 27).

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. 27).

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. 27).

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

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 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 Using the AWS SDKs (p. 20) .
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. 20).
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. 21).
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. 22).
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. 22).
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. 23).
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. 23).
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

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:

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

```
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);
```

It is recommended that 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.

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

```
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:

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

```
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:

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

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

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

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

.NET

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

            try
            {
                var describeDatabaseRequest = new DescribeDatabaseRequest
                {
                    DatabaseName = Constants.DATABASE_NAME
                };
                DescribeDatabaseResponse response = await
 writeClient.DescribeDatabaseAsync(describeDatabaseRequest);
                Console.WriteLine($"Database {Constants.DATABASE_NAME} has id:
{response.Database.Arn}");
```

```
            }
            catch (ResourceNotFoundException)
            {
                Console.WriteLine("Database does not exist.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Describe database failed:" + e.ToString());
            }

        }
```

Update database

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

Java

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

Java v2

```
public void updateDatabase(String kmsKeyId) {

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

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

        UpdateDatabaseRequest request = UpdateDatabaseRequest.builder()
                .databaseName(DATABASE_NAME)
                .kmsKeyId(kmsKeyId)
                .build();
        try {
            timestreamWriteClient.updateDatabase(request);
            System.out.println("Database [" + DATABASE_NAME + "] updated successfully
 with kmsKeyId " + kmsKeyId);
        } catch (ResourceNotFoundException e) {
```

```
            System.out.println("Database [" + DATABASE_NAME + "] does not exist.
 Skipping UpdateDatabase");
        } catch (Exception e) {
            System.out.println("UpdateDatabase failed: " + e);
        }
    }
```

Go

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

        updateDatabaseOutput, err := writeSvc.UpdateDatabase(updateDatabaseInput)

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

Python

```
def update_database(self, kmsId):
    print("Updating database")
    try:
        result = self.client.update_database(DatabaseName=Constant.DATABASE_NAME,
 KmsKeyId = kmsId)
        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

```
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:

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

```
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:

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

```
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

You can use the following code snippet to create a table:

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

```
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());
            }

        }
```

Describe table

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

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

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

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

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

```
        }
    );
}
```

.NET

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

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

        }
```

Update table

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

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

```
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:

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

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

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

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

.NET

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

List tables

You can use the following code snippets to list tables:

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

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

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

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

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

.NET

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

            try
            {
                var listTablesRequest = new ListTablesRequest
                {
                    MaxResults = 5,
                    DatabaseName = Constants.DATABASE_NAME
                };
                ListTablesResponse response = await
 writeClient.ListTablesAsync(listTablesRequest);
                PrintTables(response.Tables);
                string nextToken = response.NextToken;
                while (nextToken != null)
                {
                    listTablesRequest.NextToken = nextToken;
```

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

        }

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

Write data (inserts and upserts)

Topics

Writing batches of records

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. 206) for more information.

Java

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

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

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

        Record cpuUtilization = new Record()
                .withDimensions(dimensions)
                .withMeasureName("cpu_utilization")
                .withMeasureValue("13.5")
                .withMeasureValueType(MeasureValueType.DOUBLE)
```

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

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

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

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

Java v2

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

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

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

        Record cpuUtilization = Record.builder()
                .dimensions(dimensions)
                .measureValueType(MeasureValueType.DOUBLE)
                .measureName("cpu_utilization")
                .measureValue("13.5")
                .time(String.valueOf(time)).build();

        Record memoryUtilization = Record.builder()
                .dimensions(dimensions)
                .measureValueType(MeasureValueType.DOUBLE)
                .measureName("memory_utilization")
                .measureValue("40")
```

```
                .time(String.valueOf(time)).build();

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

        WriteRecordsRequest writeRecordsRequest = WriteRecordsRequest.builder()

.databaseName(DATABASE_NAME).tableName(TABLE_NAME).records(records).build();

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

Go

```
now := time.Now()
currentTimeInSeconds := now.Unix()
writeRecordsInput := &timestreamwrite.WriteRecordsInput{
    DatabaseName: aws.String(*databaseName),
    TableName:    aws.String(*tableName),
    Records: []*timestreamwrite.Record{
        &timestreamwrite.Record{
            Dimensions: []*timestreamwrite.Dimension{
                &timestreamwrite.Dimension{
                    Name:  aws.String("region"),
                    Value: aws.String("us-east-1"),
                },
                &timestreamwrite.Dimension{
                    Name:  aws.String("az"),
                    Value: aws.String("az1"),
                },
                &timestreamwrite.Dimension{
                    Name:  aws.String("hostname"),
                    Value: aws.String("host1"),
                },
            },
            MeasureName:      aws.String("cpu_utilization"),
            MeasureValue:     aws.String("13.5"),
            MeasureValueType: aws.String("DOUBLE"),
            Time:             aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
            TimeUnit:    aws.String("SECONDS"),
        },
        &timestreamwrite.Record{
            Dimensions: []*timestreamwrite.Dimension{
                &timestreamwrite.Dimension{
                    Name:  aws.String("region"),
                    Value: aws.String("us-east-1"),
                },
                &timestreamwrite.Dimension{
                    Name:  aws.String("az"),
                    Value: aws.String("az1"),
```

```
                },
                &timestreamwrite.Dimension{
                    Name:  aws.String("hostname"),
                    Value: aws.String("host1"),
                },
            },
            MeasureName:      aws.String("memory_utilization"),
            MeasureValue:     aws.String("40"),
            MeasureValueType: aws.String("DOUBLE"),
            Time:             aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
            TimeUnit:    aws.String("SECONDS"),
        },
    },
}

_, err = writeSvc.WriteRecords(writeRecordsInput)

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

Python

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

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

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

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

        records = [cpu_utilization, memory_utilization]

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

```
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. 206).

Java

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

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

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

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

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

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

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

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

Java v2

```
public void writeRecordsWithCommonAttributes() {
```

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

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

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

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

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

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

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

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

Go

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

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

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesInput)

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

Python

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

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

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

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

        memory_utilization = {
            'MeasureName': 'memory_utilization',
            'MeasureValue': '40'
        }
```

```
        records = [cpu_utilization, memory_utilization]

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

```
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:

Java

```
public void writeRecordsWithUpsert() {
        System.out.println("Writing records with upsert");
        // Specify repeated values for all records
        List<Record> records = new ArrayList<>();
        final long time = System.currentTimeMillis();
        // To achieve upsert (last writer wins) semantic, one example is to use current
 time as the version if you are writing directly from the data source
        long version = System.currentTimeMillis();

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

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

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

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

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

        WriteRecordsRequest writeRecordsRequest = new WriteRecordsRequest()
```

```
                .withDatabaseName(DATABASE_NAME)
                .withTableName(TABLE_NAME)
                .withCommonAttributes(commonAttributes);
        writeRecordsRequest.setRecords(records);

        // write records for first time
        try {
            WriteRecordsResult writeRecordsResult =
amazonTimestreamWrite.writeRecords(writeRecordsRequest);
            System.out.println("WriteRecords Status for first time: " +
writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
        } catch (RejectedRecordsException e) {
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }

        // Successfully retry same writeRecordsRequest with same records and versions,
because writeRecords API is idempotent.
        try {
            WriteRecordsResult writeRecordsResult =
amazonTimestreamWrite.writeRecords(writeRecordsRequest);
            System.out.println("WriteRecords Status for retry: " +
writeRecordsResult.getSdkHttpMetadata().getHttpStatusCode());
        } catch (RejectedRecordsException e) {
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }

        // upsert with lower version, this would fail because a higher version is
required to update the measure value.
        version -= 1;
        commonAttributes.setVersion(version);

        cpuUtilization.setMeasureValue("14.5");
        memoryUtilization.setMeasureValue("50");

        List<Record> upsertedRecords = new ArrayList<>();
        upsertedRecords.add(cpuUtilization);
        upsertedRecords.add(memoryUtilization);

        WriteRecordsRequest writeRecordsUpsertRequest = new WriteRecordsRequest()
                .withDatabaseName(DATABASE_NAME)
                .withTableName(TABLE_NAME)
                .withCommonAttributes(commonAttributes);
        writeRecordsUpsertRequest.setRecords(upsertedRecords);

        try {
            WriteRecordsResult writeRecordsUpsertResult =
amazonTimestreamWrite.writeRecords(writeRecordsUpsertRequest);
            System.out.println("WriteRecords Status for upsert with lower version: " +
writeRecordsUpsertResult.getSdkHttpMetadata().getHttpStatusCode());
        } catch (RejectedRecordsException e) {
            System.out.println("WriteRecords Status for upsert with lower version: ");
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }

        // upsert with higher version as new data in generated
        version = System.currentTimeMillis();
        commonAttributes.setVersion(version);

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

```
                .withTableName(TABLE_NAME)
                .withCommonAttributes(commonAttributes);
        writeRecordsUpsertRequest.setRecords(upsertedRecords);

        try {
            WriteRecordsResult writeRecordsUpsertResult =
 amazonTimestreamWrite.writeRecords(writeRecordsUpsertRequest);
            System.out.println("WriteRecords Status for upsert with higher version: " +
 writeRecordsUpsertResult.getSdkHttpMetadata().getHttpStatusCode());
        } catch (RejectedRecordsException e) {
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
```

Java v2

```
public void writeRecordsWithUpsert() {
        System.out.println("Writing records with upsert");
        // Specify repeated values for all records
        List<Record> records = new ArrayList<>();
        final long time = System.currentTimeMillis();
        // To achieve upsert (last writer wins) semantic, one example is to use current
 time as the version if you are writing directly from the data source
        long version = System.currentTimeMillis();

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

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

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

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

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

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

        // write records for first time
        try {
            WriteRecordsResponse writeRecordsResponse =
 timestreamWriteClient.writeRecords(writeRecordsRequest);
```

```
            System.out.println("WriteRecords Status for first time: " +
writeRecordsResponse.sdkHttpResponse().statusCode());
        } catch (RejectedRecordsException e) {
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }

        // Successfully retry same writeRecordsRequest with same records and versions,
because writeRecords API is idempotent.
        try {
            WriteRecordsResponse writeRecordsResponse =
timestreamWriteClient.writeRecords(writeRecordsRequest);
            System.out.println("WriteRecords Status for retry: " +
writeRecordsResponse.sdkHttpResponse().statusCode());
        } catch (RejectedRecordsException e) {
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }

        // upsert with lower version, this would fail because a higher version is
required to update the measure value.
        version -= 1;
        commonAttributes = Record.builder()
                .dimensions(dimensions)
                .measureValueType(MeasureValueType.DOUBLE)
                .time(String.valueOf(time))
                .version(version)
                .build();

        cpuUtilization = Record.builder()
                .measureName("cpu_utilization")
                .measureValue("14.5").build();
        memoryUtilization = Record.builder()
                .measureName("memory_utilization")
                .measureValue("50").build();

        List<Record> upsertedRecords = new ArrayList<>();
        upsertedRecords.add(cpuUtilization);
        upsertedRecords.add(memoryUtilization);

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

        try {
            WriteRecordsResponse writeRecordsResponse =
timestreamWriteClient.writeRecords(writeRecordsUpsertRequest);
            System.out.println("WriteRecords Status for upsert with lower version: " +
writeRecordsResponse.sdkHttpResponse().statusCode());
        } catch (RejectedRecordsException e) {
            System.out.println("WriteRecords Status for upsert with lower version: ");
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }

        // upsert with higher version as new data in generated
        version = System.currentTimeMillis();
        commonAttributes = Record.builder()
                .dimensions(dimensions)
                .measureValueType(MeasureValueType.DOUBLE)
                .time(String.valueOf(time))
```

```
                .version(version)
                .build();

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

        try {
            WriteRecordsResponse writeRecordsUpsertResponse =
 timestreamWriteClient.writeRecords(writeRecordsUpsertRequest);
            System.out.println("WriteRecords Status for upsert with higher version: " +
 writeRecordsUpsertResponse.sdkHttpResponse().statusCode());
        } catch (RejectedRecordsException e) {
            printRejectedRecordsException(e);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
```

Go

```
// Below code will ingest and upsert cpu_utilization and memory_utilization metric for
 a host on
// region=us-east-1, az=az1, and hostname=host1
fmt.Println("Ingesting records and set version as currentTimeInMills, hit enter to
 continue")
reader.ReadString('\n')

// Get current time in seconds.
now = time.Now()
currentTimeInSeconds = now.Unix()
// To achieve upsert (last writer wins) semantic, one example is to use current time as
 the version if you are writing directly from the data source
version := time.Now().Round(time.Millisecond).UnixNano() / 1e6   // set version as
 currentTimeInMills

writeRecordsCommonAttributesUpsertInput := &timestreamwrite.WriteRecordsInput{
 DatabaseName: aws.String(*databaseName),
 TableName:    aws.String(*tableName),
 CommonAttributes: &timestreamwrite.Record{
  Dimensions: []*timestreamwrite.Dimension{
   &timestreamwrite.Dimension{
    Name:  aws.String("region"),
    Value: aws.String("us-east-1"),
   },
   &timestreamwrite.Dimension{
    Name:  aws.String("az"),
    Value: aws.String("az1"),
   },
   &timestreamwrite.Dimension{
    Name:  aws.String("hostname"),
    Value: aws.String("host1"),
   },
  },
  MeasureValueType: aws.String("DOUBLE"),
  Time:             aws.String(strconv.FormatInt(currentTimeInSeconds, 10)),
  TimeUnit:    aws.String("SECONDS"),
  Version:          &version,
 },
 Records: []*timestreamwrite.Record{
  &timestreamwrite.Record{
   MeasureName:  aws.String("cpu_utilization"),
   MeasureValue: aws.String("13.5"),
```

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

// write records for first time
_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

if err != nil {
 fmt.Println("Error:")
 fmt.Println(err)
} else {
 fmt.Println("Frist-time write records is successful")
}

fmt.Println("Retry same writeRecordsRequest with same records and versions. Because
 writeRecords API is idempotent, this will success. hit enter to continue")
reader.ReadString('\n')

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

if err != nil {
 fmt.Println("Error:")
 fmt.Println(err)
} else {
 fmt.Println("Retry write records for same request is successful")
}

fmt.Println("Upsert with lower version, this would fail because a higher version is
 required to update the measure value. hit enter to continue")
reader.ReadString('\n')
version -= 1
writeRecordsCommonAttributesUpsertInput.CommonAttributes.Version = &version

updated_cpu_utilization := &timestreamwrite.Record{
 MeasureName:      aws.String("cpu_utilization"),
 MeasureValue:     aws.String("14.5"),
}
updated_memory_utilization := &timestreamwrite.Record{
 MeasureName:      aws.String("memory_utilization"),
 MeasureValue:     aws.String("50"),
}


writeRecordsCommonAttributesUpsertInput.Records = []*timestreamwrite.Record{
 updated_cpu_utilization,
 updated_memory_utilization,
}

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

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

fmt.Println("Upsert with higher version as new data in generated, this would success.
 hit enter to continue")
reader.ReadString('\n')
```

```
version = time.Now().Round(time.Millisecond).UnixNano() / 1e6  // set version as
 currentTimeInMills
writeRecordsCommonAttributesUpsertInput.CommonAttributes.Version = &version

_, err = writeSvc.WriteRecords(writeRecordsCommonAttributesUpsertInput)

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

Python

```
def write_records_with_upsert(self):
        print("Writing records with upsert")
        current_time = self._current_milli_time()
        # To achieve upsert (last writer wins) semantic, one example is to use current
 time as the version if you are writing directly from the data source
        version = int(self._current_milli_time())

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

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

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

        memory_utilization = {
            'MeasureName': 'memory_utilization',
            'MeasureValue': '40'
        }

        records = [cpu_utilization, memory_utilization]

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

        # Successfully retry same writeRecordsRequest with same records and versions,
 because writeRecords API is idempotent.
        try:
            result = self.client.write_records(DatabaseName=Constant.DATABASE_NAME,
 TableName=Constant.TABLE_NAME,
```

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

        # upsert with lower version, this would fail because a higher version is
 required to update the measure value.
        version -= 1
        common_attributes["Version"] = version

        cpu_utilization["MeasureValue"] = '14.5'
        memory_utilization["MeasureValue"] = '50'

        upsertedRecords = [cpu_utilization, memory_utilization]

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


        # upsert with higher version as new data in 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 in generated
        now = DateTimeOffset.UtcNow;
        version = now.ToUnixTimeMilliseconds();
        recordType.GetProperty("Version").SetValue(commonAttributes, version);
```

```
        try
        {
            var writeRecordsUpsertRequest = new WriteRecordsRequest
            {
                DatabaseName = Constants.DATABASE_NAME,
                TableName = Constants.TABLE_NAME,
                Records = upsertedRecords,
                CommonAttributes = commonAttributes
            };
            WriteRecordsResponse upsertResponse = await
writeClient.WriteRecordsAsync(writeRecordsUpsertRequest);
            Console.WriteLine($"WriteRecords Status for upsert with higher version:
{upsertResponse.HttpStatusCode.ToString()}");
        }
        catch (RejectedRecordsException e) {
            PrintRejectedRecordsException(e);
        }
        catch (Exception e)
        {
            Console.WriteLine("Write records failure:" + e.ToString());
        }
    }
```

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.

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

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.

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)
            traceback.print_exc(file=sys.stderr)
```

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.

Java

```
private void parseQueryResult(QueryResult response) {
        final QueryStatus currentStatusOfQuery = queryResult.getQueryStatus();

        System.out.println("Query progress so far: " +
 currentStatusOfQuery.getProgressPercentage() + "%");

        double bytesScannedSoFar = ((double)
 currentStatusOfQuery.getCumulativeBytesScanned() / ONE_GB_IN_BYTES);
        System.out.println("Bytes scanned so far: " + bytesScannedSoFar + " GB");

        double bytesMeteredSoFar = ((double)
 currentStatusOfQuery.getCumulativeBytesMetered() / ONE_GB_IN_BYTES);
        System.out.println("Bytes metered so far: " + bytesMeteredSoFar + " GB");

        List<ColumnInfo> columnInfo = response.getColumnInfo();
        List<Row> rows = response.getRows();

        System.out.println("Metadata: " + columnInfo);
        System.out.println("Data: ");

        // iterate every row
        for (Row row : rows) {
            System.out.println(parseRow(columnInfo, row));
        }
    }

    private String parseRow(List<ColumnInfo> columnInfo, Row row) {
        List<Datum> data = row.getData();
        List<String> rowOutput = new ArrayList<>();
        // iterate every column per row
        for (int j = 0; j < data.size(); j++) {
            ColumnInfo info = columnInfo.get(j);
            Datum datum = data.get(j);
            rowOutput.add(parseDatum(info, datum));
        }
        return String.format("{%s}",
 rowOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }
```

```
   private String parseDatum(ColumnInfo info, Datum datum) {
       if (datum.isNullValue() != null && datum.isNullValue()) {
           return info.getName() + "=" + "NULL";
       }
       Type columnType = info.getType();
       // If the column is of TimeSeries Type
       if (columnType.getTimeSeriesMeasureValueColumnInfo() != null) {
           return parseTimeSeries(info, datum);
       }
       // If the column is of Array Type
       else if (columnType.getArrayColumnInfo() != null) {
           List<Datum> arrayValues = datum.getArrayValue();
           return info.getName() + "=" +
parseArray(info.getType().getArrayColumnInfo(), arrayValues);
       }
       // If the column is of Row Type
       else if (columnType.getRowColumnInfo() != null) {
           List<ColumnInfo> rowColumnInfo = info.getType().getRowColumnInfo();
           Row rowValues = datum.getRowValue();
           return parseRow(rowColumnInfo, rowValues);
       }
       // If the column is of Scalar Type
       else {
           return parseScalarType(info, datum);
       }
   }

   private String parseTimeSeries(ColumnInfo info, Datum datum) {
       List<String> timeSeriesOutput = new ArrayList<>();
       for (TimeSeriesDataPoint dataPoint : datum.getTimeSeriesValue()) {
           timeSeriesOutput.add("{time=" + dataPoint.getTime() + ", value=" +
                   parseDatum(info.getType().getTimeSeriesMeasureValueColumnInfo(),
dataPoint.getValue()) + "}");
       }
       return String.format("[%s]",
timeSeriesOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
   }

   private String parseScalarType(ColumnInfo info, Datum datum) {
       switch (ScalarType.fromValue(info.getType().getScalarType())) {
           case VARCHAR:
               return parseColumnName(info) + datum.getScalarValue();
           case BIGINT:
               Long longValue = Long.valueOf(datum.getScalarValue());
               return parseColumnName(info) + longValue;
           case INTEGER:
               Integer intValue = Integer.valueOf(datum.getScalarValue());
               return parseColumnName(info) + intValue;
           case BOOLEAN:
               Boolean booleanValue = Boolean.valueOf(datum.getScalarValue());
               return parseColumnName(info) + booleanValue;
           case DOUBLE:
               Double doubleValue = Double.valueOf(datum.getScalarValue());
               return parseColumnName(info) + doubleValue;
           case TIMESTAMP:
               return parseColumnName(info) +
LocalDateTime.parse(datum.getScalarValue(), TIMESTAMP_FORMATTER);
           case DATE:
               return parseColumnName(info) + LocalDate.parse(datum.getScalarValue(),
DATE_FORMATTER);
           case TIME:
               return parseColumnName(info) + LocalTime.parse(datum.getScalarValue(),
TIME_FORMATTER);
           case INTERVAL_DAY_TO_SECOND:
           case INTERVAL_YEAR_TO_MONTH:
               return parseColumnName(info) + datum.getScalarValue();
```

```
            case UNKNOWN:
                return parseColumnName(info) + datum.getScalarValue();
            default:
                throw new IllegalArgumentException("Given type is not valid: " +
 info.getType().getScalarType());
        }
    }

    private String parseColumnName(ColumnInfo info) {
        return info.getName() == null ? "" : info.getName() + "=";
    }

    private String parseArray(ColumnInfo arrayColumnInfo, List<Datum> arrayValues) {
        List<String> arrayOutput = new ArrayList<>();
        for (Datum datum : arrayValues) {
            arrayOutput.add(parseDatum(arrayColumnInfo, datum));
        }
        return String.format("[%s]",
 arrayOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }
```

Java v2

```
private void parseQueryResult(QueryResponse response) {
        final QueryStatus currentStatusOfQuery = response.queryStatus();

        System.out.println("Query progress so far: " +
 currentStatusOfQuery.progressPercentage() + "%");

        double bytesScannedSoFar = ((double)
 currentStatusOfQuery.cumulativeBytesScanned() / ONE_GB_IN_BYTES);
        System.out.println("Bytes scanned so far: " + bytesScannedSoFar + " GB");

        double bytesMeteredSoFar = ((double)
 currentStatusOfQuery.cumulativeBytesMetered() / ONE_GB_IN_BYTES);
        System.out.println("Bytes metered so far: " + bytesMeteredSoFar + " GB");

        List<ColumnInfo> columnInfo = response.columnInfo();
        List<Row> rows = response.rows();

        System.out.println("Metadata: " + columnInfo);
        System.out.println("Data: ");

        // iterate every row
        for (Row row : rows) {
            System.out.println(parseRow(columnInfo, row));
        }
    }

    private String parseRow(List<ColumnInfo> columnInfo, Row row) {
        List<Datum> data = row.data();
        List<String> rowOutput = new ArrayList<>();
        // iterate every column per row
        for (int j = 0; j < data.size(); j++) {
            ColumnInfo info = columnInfo.get(j);
            Datum datum = data.get(j);
            rowOutput.add(parseDatum(info, datum));
        }
        return String.format("{%s}",
 rowOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }

    private String parseDatum(ColumnInfo info, Datum datum) {
        if (datum.nullValue() != null && datum.nullValue()) {
            return info.name() + "=" + "NULL";
```

```
        }
        Type columnType = info.type();
        // If the column is of TimeSeries Type
        if (columnType.timeSeriesMeasureValueColumnInfo() != null) {
            return parseTimeSeries(info, datum);
        }
        // If the column is of Array Type
        else if (columnType.arrayColumnInfo() != null) {
            List<Datum> arrayValues = datum.arrayValue();
            return info.name() + "=" + parseArray(info.type().arrayColumnInfo(),
arrayValues);
        }
        // If the column is of Row Type
        else if (columnType.rowColumnInfo() != null &&
columnType.rowColumnInfo().size() > 0) {
            List<ColumnInfo> rowColumnInfo = info.type().rowColumnInfo();
            Row rowValues = datum.rowValue();
            return parseRow(rowColumnInfo, rowValues);
        }
        // If the column is of Scalar Type
        else {
            return parseScalarType(info, datum);
        }
    }

    private String parseTimeSeries(ColumnInfo info, Datum datum) {
        List<String> timeSeriesOutput = new ArrayList<>();
        for (TimeSeriesDataPoint dataPoint : datum.timeSeriesValue()) {
            timeSeriesOutput.add("{time=" + dataPoint.time() + ", value=" +
                    parseDatum(info.type().timeSeriesMeasureValueColumnInfo(),
dataPoint.value()) + "}");
        }
        return String.format("[%s]",
timeSeriesOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }

    private String parseScalarType(ColumnInfo info, Datum datum) {
        return parseColumnName(info) + datum.scalarValue();
    }

    private String parseColumnName(ColumnInfo info) {
        return info.name() == null ? "" : info.name() + "=";
    }

    private String parseArray(ColumnInfo arrayColumnInfo, List<Datum> arrayValues) {
        List<String> arrayOutput = new ArrayList<>();
        for (Datum datum : arrayValues) {
            arrayOutput.add(parseDatum(arrayColumnInfo, datum));
        }
        return String.format("[%s]",
arrayOutput.stream().map(Object::toString).collect(Collectors.joining(",")));
    }
```

Go

```
func processScalarType(data *timestreamquery.Datum) string {
    return *data.ScalarValue
}

func processTimeSeriesType(data []*timestreamquery.TimeSeriesDataPoint, columnInfo
 *timestreamquery.ColumnInfo) string {
    value := ""
    for k := 0; k < len(data); k++ {
        time := data[k].Time
        value += *time + ":"
```

```
        if columnInfo.Type.ScalarType != nil {
            value += processScalarType(data[k].Value)
        } else if columnInfo.Type.ArrayColumnInfo != nil {
            value += processArrayType(data[k].Value.ArrayValue,
 columnInfo.Type.ArrayColumnInfo)
        } else if columnInfo.Type.RowColumnInfo != nil {
            value += processRowType(data[k].Value.RowValue.Data,
 columnInfo.Type.RowColumnInfo)
        } else {
            fail("Bad data type")
        }
        if k != len(data)-1 {
            value += ", "
        }
    }
    return value
}

func processArrayType(datumList []*timestreamquery.Datum, columnInfo
 *timestreamquery.ColumnInfo) string {
    value := ""
    for k := 0; k < len(datumList); k++ {
        if columnInfo.Type.ScalarType != nil {
            value += processScalarType(datumList[k])
        } else if columnInfo.Type.TimeSeriesMeasureValueColumnInfo != nil {
            value += processTimeSeriesType(datumList[k].TimeSeriesValue,
 columnInfo.Type.TimeSeriesMeasureValueColumnInfo)
        } else if columnInfo.Type.ArrayColumnInfo != nil {
            value += "["
            value += processArrayType(datumList[k].ArrayValue,
 columnInfo.Type.ArrayColumnInfo)
            value += "]"
        } else if columnInfo.Type.RowColumnInfo != nil {
            value += "["
            value += processRowType(datumList[k].RowValue.Data,
 columnInfo.Type.RowColumnInfo)
            value += "]"
        } else {
            fail("Bad column type")
        }

        if k != len(datumList)-1 {
            value += ", "
        }
    }
    return value
}

func processRowType(data []*timestreamquery.Datum, metadata
 []*timestreamquery.ColumnInfo) string {
    value := ""
    for j := 0; j < len(data); j++ {
        if metadata[j].Type.ScalarType != nil {
            // process simple data types
            value += processScalarType(data[j])
        } else if metadata[j].Type.TimeSeriesMeasureValueColumnInfo != nil {
            // fmt.Println("Timeseries measure value column info")
            // fmt.Println(metadata[j].Type.TimeSeriesMeasureValueColumnInfo.Type)
            datapointList := data[j].TimeSeriesValue
            value += "["
            value += processTimeSeriesType(datapointList,
 metadata[j].Type.TimeSeriesMeasureValueColumnInfo)
            value += "]"
        } else if metadata[j].Type.ArrayColumnInfo != nil {
            columnInfo := metadata[j].Type.ArrayColumnInfo
            // fmt.Println("Array column info")
```

```
            // fmt.Println(columnInfo)
            datumList := data[j].ArrayValue
            value += "["
            value += processArrayType(datumList, columnInfo)
            value += "]"
        } else if metadata[j].Type.RowColumnInfo != nil {
            columnInfo := metadata[j].Type.RowColumnInfo
            datumList := data[j].RowValue.Data
            value += "["
            value += processRowType(datumList, columnInfo)
            value += "]"
        } else {
            panic("Bad column type")
        }
        // comma seperated column values
        if j != len(data)-1 {
            value += ", "
        }
    }
    return value
}
```

Python

```
def __parse_query_result(self, query_result):
        query_status = query_result["QueryStatus"]

        progress_percentage = query_status["ProgressPercentage"]
        print("Query progress so far: " + str(progress_percentage) + "%")

        bytes_scanned = query_status["CumulativeBytesScanned"] / self.ONE_GB_IN_BYTES
        print("Bytes Scanned so far: " + str(bytes_scanned) + " GB")

        bytes_metered = query_status["CumulativeBytesMetered"] / self.ONE_GB_IN_BYTES
        print("Bytes Metered so far: " + str(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_coulmn_info = info['Type']['RowColumnInfo']
            row_values = datum['RowValue']
            return self.__parse_row(row_coulmn_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_column_name(self, info):
        if 'Name' in info:
            return info['Name'] + "="
        else:
            return ""

    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)
```

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.

Java

```
private static final long ONE_GB_IN_BYTES = 1073741824L;
private static final double QUERY_COST_PER_GB_IN_DOLLARS = 0.01; // Assuming the price
 of query is $0.01 per GB

public void cancelQueryBasedOnQueryStatus() {
        System.out.println("Starting query: " + SELECT_ALL_QUERY);
        QueryRequest queryRequest = new QueryRequest();
        queryRequest.setQueryString(SELECT_ALL_QUERY);
        QueryResult queryResult = queryClient.query(queryRequest);

        while (true) {
            final QueryStatus currentStatusOfQuery = queryResult.getQueryStatus();
            System.out.println("Query progress so far: " +
 currentStatusOfQuery.getProgressPercentage() + "%");
            double bytesMeteredSoFar = ((double)
 currentStatusOfQuery.getCumulativeBytesMetered() / ONE_GB_IN_BYTES);
            System.out.println("Bytes metered so far: " + bytesMeteredSoFar + " GB");
            // Cancel query if its costing more than 1 cent
            if (bytesMeteredSoFar * QUERY_COST_PER_GB_IN_DOLLARS > 0.01) {
                cancelQuery(queryResult);
                break;
            }

            if (queryResult.getNextToken() == null) {
                break;
            }
            queryRequest.setNextToken(queryResult.getNextToken());
            queryResult = queryClient.query(queryRequest);
```

```
        }
    }
```

Java v2

```
private static final long ONE_GB_IN_BYTES = 1073741824L;
private static final double QUERY_COST_PER_GB_IN_DOLLARS = 0.01; // Assuming the price
 of query is $0.01 per GB

public void cancelQueryBasedOnQueryStatus() {
        System.out.println("Starting query: " + SELECT_ALL_QUERY);
        QueryRequest queryRequest =
 QueryRequest.builder().queryString(SELECT_ALL_QUERY).build();

        final QueryIterable queryResponseIterator =
 timestreamQueryClient.queryPaginator(queryRequest);
        for(QueryResponse queryResponse : queryResponseIterator) {
            final QueryStatus currentStatusOfQuery = queryResponse.queryStatus();
            System.out.println("Query progress so far: " +
 currentStatusOfQuery.progressPercentage() + "%");
            double bytesMeteredSoFar = ((double)
 currentStatusOfQuery.cumulativeBytesMetered() / ONE_GB_IN_BYTES);
            System.out.println("Bytes metered so far: " + bytesMeteredSoFar + "GB");
            // Cancel query if its costing more than 1 cent
            if (bytesMeteredSoFar * QUERY_COST_PER_GB_IN_DOLLARS > 0.01) {
                cancelQuery(queryResponse);
                break;
            }
        }
    }
```

Go

```
const OneGbInBytes = 1073741824
// Assuming the price of query is $0.01 per GB
const QueryCostPerGbInDollars = 0.01

func cancelQueryBasedOnQueryStatus(queryPtr *string, querySvc
 *timestreamquery.TimestreamQuery, f *os.File) {
    queryInput := &timestreamquery.QueryInput{
        QueryString: aws.String(*queryPtr),
    }
    fmt.Println("QueryInput:")
    fmt.Println(queryInput)
    // execute the query
    err := querySvc.QueryPages(queryInput,
        func(page *timestreamquery.QueryOutput, lastPage bool) bool {
            // process query response
            queryStatus := page.QueryStatus
            fmt.Println("Current query status:", queryStatus)
            bytes_metered := float64(*queryStatus.CumulativeBytesMetered) /
 float64(ONE_GB_IN_BYTES)
            if bytes_metered * QUERY_COST_PER_GB_IN_DOLLARS > 0.01 {
                cancelQuery(page, querySvc)
                return true
            }
            // query response metadata
            // includes column names and types
            metadata := page.ColumnInfo
            // fmt.Println("Metadata:")
            fmt.Println(metadata)
            header := ""
            for i := 0; i < len(metadata); i++ {
                header += *metadata[i].Name
```

```
                if i != len(metadata)-1 {
                    header += ", "
                }
            }
            write(f, header)

            // query response data
            fmt.Println("Data:")
            // process rows
            rows := page.Rows
            for i := 0; i < len(rows); i++ {
                data := rows[i].Data
                value := processRowType(data, metadata)
                fmt.Println(value)
                write(f, value)
            }
            fmt.Println("Number of rows:", len(page.Rows))
            return true
        })
    if err != nil {
        fmt.Println("Error:")
        fmt.Println(err)
    }
}
```

Python

```
ONE_GB_IN_BYTES = 1073741824
# Assuming the price of query is $0.01 per GB
QUERY_COST_PER_GB_IN_DOLLARS = 0.01

def cancel_query_based_on_query_status(self):
        try:
            print("Starting query: " + self.SELECT_ALL)
            page_iterator = self.paginator.paginate(QueryString=self.SELECT_ALL)
            for page in page_iterator:
                query_status = page["QueryStatus"]
                progress_percentage = query_status["ProgressPercentage"]
                print("Query progress so far: " + str(progress_percentage) + "%")
                bytes_metered = query_status["CumulativeBytesMetered"] /
 self.ONE_GB_IN_BYTES
                print("Bytes Metered so far: " + str(bytes_metered) + " GB")
                if bytes_metered * self.QUERY_COST_PER_GB_IN_DOLLARS > 0.01:
                    self.cancel_query_for(page)
                    break
        except Exception as err:
            print("Exception while running query:", err)
            traceback.print_exc(file=sys.stderr)
```

Node.js

```
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. 93).

Cancel query

You can use the following code snippets to cancel a query:

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);
            }
        }
```

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. 96).
2. Create tags by using Tagging Operations (p. 96).

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

You can use the Timestream console to add tags to new databases and tables 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 existing resources (console)

1. Open the Timestream console at https://console.aws.amazon.com/timestream.
2. In the navigation pane, choose **Databases** or **Tables**.
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. 96).

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

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

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. 100).

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. 100) 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:

- *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.

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

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. 119).

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 employees 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. 104).

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. 110).

Authenticating with Identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see Signing in to the AWS Management Console as an IAM user or root user in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also 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 Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you first create an AWS account, you begin with a single 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, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM Users and Groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

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:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated users and roles in the *IAM User Guide*.
- **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 IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. 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. In other words, 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

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. 106)	Read	table*
CancelQuery	Cancel a query.	Read	*
ListTables	Get the list of tables.	List	database*
ListDatabases	Get the list of databases.	List	*
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	Read	*

Actions	Description	Access Level	Resource Types (*required)
	be specified. See this note for a detailed explanation (p. 106).		
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, specifcially 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. 110).

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. 96).

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. 117).

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
AmazonTimestreamConsoleFullAccess – Update to an existing policy	Timestream removed redundant actions from the existing managed policy AmazonTimestreamConsoleFullAccess (p. 12). Previously, the AmazonTimestreamConsoleFullAccess managed policy included a redundant action, *dbqms:DescribeQueryHistory*. The old policy can be viewed below: `"Action": ["dbqms:CreateFavoriteQuery", "dbqms:DescribeFavoriteQueries", "dbqms:UpdateFavoriteQuery", "dbqms:DeleteFavoriteQueries", "dbqms:GetQueryString", "dbqms:CreateQueryHistory", "dbqms:DescribeQueryHistory", "dbqms:UpdateQueryHistory", "dbqms:DeleteQueryHistory", "dbqms:DescribeQueryHistory"]` 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. `"Action": ["dbqms:CreateFavoriteQuery", "dbqms:DescribeFavoriteQueries", "dbqms:UpdateFavoriteQuery", "dbqms:DeleteFavoriteQueries", "dbqms:GetQueryString", "dbqms:CreateQueryHistory", "dbqms:UpdateQueryHistory",`	April 23, 2021

Change	Description	Date
	"dbqms:DeleteQueryHistory", "dbqms:DescribeQueryHistory"]	
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

Identity-based policies are very powerful. They 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 using AWS managed policies** – To start using Timestream quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get started using permissions with AWS managed policies in the *IAM User Guide*.
- **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant least privilege in the *IAM User Guide*.
- **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.
- **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON policy elements: Condition 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

- Create, Describe, Delete and Describe a Database (p. 115)
- Limit Listed Databases by Tag{"Owner": "${username}"} (p. 115)
- List All Tables in a Database (p. 116)
- Create, Describe, Delete, Update and Select on a Table (p. 116)
- Limit a Query by Table (p. 116)

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.

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

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, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies 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 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 asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

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. 104).
- 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

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

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. 122).

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?

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 `SystemErrors` to determine whether any requests resulted in a server error code. Typically, this metric should be equal to zero. If it isn't, you might want to 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

Amazon CloudWatch aggregates the following Timestream metrics:

> **Note**
> All metrics are aggregated at one-minute intervals.

Metric	Description
`SystemErrors`	The requests to Timestream that generate a SystemError during the specified time period. A SystemError usually indicates an internal service error. Units: `Count` Dimensions: `Operation` Valid Statistics: • `Sum` • `SampleCount`
`UserErrors`	Requests to Timestream that generate an InvalidRequest error during the specified time period. An InvalidRequest usually indicates a client-side error, such as an invalid combination of

Metric	Description
	parameters, an attempt to update a nonexistent table, or an incorrect request signature. UserErrors represents the aggregate of invalid requests for the current AWS Region and the current AWS account. Units: `Count` Dimensions: `Operation` Valid Statistics: • `Sum` • `SampleCount`
`SuccessfulRequestLatency`	The successful requests to Timestream during the specified time period. SuccessfulRequestLatency can provide two different kinds of information: • The elapsed time for successful requests (Minimum, Maximum,Sum, or Average). • The number of successful requests (SampleCount). SuccessfulRequestLatency reflects activity only within Timestream and does not take into account network latency or client-side activity. Units: `Milliseconds` **Dimensions** • `DatabaseName` • `TableName` • `Operation` Valid Statistics: • `Minimum` • `Maximum` • `Average` • `SampleCount` • `P10` • `p50` • `p90` • `p95` • `p99`

Metric	Description
`CumulativeBytesMetered`	The amount of data scanned by queries sent to Timestream, in bytes. Units: `Bytes` Dimensions: `Operation` Valid Statistics: • `Sum`

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, bu 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. 125).

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
`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 Supported SQL Constructs are logged by CloudTrail. Timestream's supported SQL constructs map to the following Timestream IAM actions, as shown below:

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, and/or Query API calls made 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. 126).

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

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 `cell1` in `us-west-2`, and you've set up VPC interface endpoints for writes (`ingest-cell1.timestream.us-west-2.amazonaws.com`) and queries (`query-cell1.timestream.us-west-2.amazonaws.com`). In this case, any write requests sent using these endpoints will stay entirely within the Amazon network and will not access the public internet.

Considerations for Timestream VPC endpoints

Consider the following when creating a VPC endpoint for Timestream:

- Before you set up an interface VPC endpoint for Timestream, 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. 129) section of this guide.

Now that you understand how Timestream works with VPC endpoints, create an interface VPC endpoint for Timestream (p. 129).

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. 128)

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. 130) 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 (`cell1`, `cell2`,etc.) returned in the Endpoints object by the DescribeEndpoints action. See the example (p. 130) 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. 130).

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

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. 107) >

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

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

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. 12).
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
```

3. Because the version of boto3 that Lambda uses does not yet include Timestream, you need to add it to the package that you will deploy to Lambda. The steps to do this are described below:

a. (optional) Create a virtual environment:

```
python3 -m venv venv
. venv/bin/activate
```

b. If your version of boto3 is older than 1.15.9, reinstall it with the following commands:

```
pip3 uninstall boto3
pip3 install boto3
```

c. Create a file named `lambda_function.py`, and copy and paste the code below:

```
import json
import os
import boto3
from botocore.config import Config

def lambda_handler(event, context):
    session = boto3.Session()
    write_client = session.client('timestream-write',
 config=Config(read_timeout=20, max_pool_connections=5000,

 retries={'max_attempts': 10}))

    result = write_client.list_databases(MaxResults=5)
    databases = result['Databases']
    for database in databases:
        print(database['DatabaseName'])

    return {
        'statusCode': 200,
        'body': json.dumps('Hello Timestream from Lambda!')
    }
```

d. Run the following commands in the same directory as the one in which you created `lambda_function.py`:

```
# Install boto3 locally
pip3 install --target ./package boto3
```

```
# Create zip with contents of the installed packages
cd package
zip -r9 ${OLDPWD}/function.zip .

# Add lambda_function.py to the package
cd $OLDPWD
zip -g function.zip lambda_function.py

# Create a function in AWS Lambda function
aws lambda create-function --function-name TimestreamExample --runtime python3.8 --
role <arn_of_the_role_from_step_1>  --handler lambda_function.lambda_handler --zip-
file fileb://function.zip
```

Build AWS Lambda functions using Amazon Timestream with JavaScript

To build AWS Lambda functions using Amazon Timestream with JavaScript, follow the instructions outlined here.

Build AWS Lambda functions using Amazon Timestream with Go

To build AWS Lambda functions using Amazon Timestream with Go, follow the instructions outlined here.

Build AWS Lambda functions using Amazon Timestream with C#

To build AWS Lambda functions using Amazon Timestream with C#, follow the instructions outlined here.

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

1. Create a database in Amazon Timestream using the instructions described in Create a database (p. 16).
2. Create a table in Amazon Timestream using the instructions described in Create a table (p. 16).

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

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. 16)
2. Create a table in Timestream with the name `kinesisdata1` following the instructions described in Create a table (p. 16)

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. 16)

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. 138) 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. 138) 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

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. 140).
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. 140).
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. 140).
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. 140).
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. 16) and Create a table (p. 16)
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

1. Create a database and a table in Timestream following the instructions described in Create a database (p. 16) 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. 22)
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. Install Grafana
7. 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 tabl
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 alerts
22. To troubleshoot alerts, follow the instructions described in the Grafana documentation to Troubleshoot alerts
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 open source Telegraf works with Timestream, how to install Telegraf with the Timestream output plugin, and how to run Telegraf with the Timestream output plugin.

Topics

Mapping Telegraf/InfluxDB Metrics to the Timestream model

The Timestream output plugin can map your Telegraf/InfluxDB metrics into Timestream in two different ways: *single-table* and *multi-table*.

> **Note**
> In most cases, using `multi-table` mapping mode is recommended. However, you can use `single-table` mode in situations when you have thousands of measurement names.

For example, consider the following 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
```

In this example, `weather` and `airquality` are the measurement names, `location` and `season` are tags, and `temperature`, `humidity`, `no2`, and `pm25` are fields.

For the data above, examples of *single-table* and *multi-table* mode are given below:

Topics

Multi-table mode

In *multi-table* mode, the following mapping will occur:

- The first line of the data will be ingested to table named `weather`
- The second line will be ingested to table named `airquality`
- The tags will be represented as dimensions
- The first table (`weather`) will have two records, one with a measurement name equal to `temperature`, and another with a measurement name equal to `humidity`.
- The second table (`airquality`) will have two records, one with a measurement name equal to `no2`, and another with a measurement name equal to `pm25`.

The resulting Timestream tables, `weather` and `airquality`, will look like this:

`weather`

Time	location	season	measure_name	measure_value::bigint
5:43 pm	us-midwest	summer	temperature	82
5:43 pm	us-midwest	summer	humidity	71

`airquality`

Time	location	measure_name	measure_value::bigint
17:43:50	us-west	no2	5
17:43:50	us-west	pm25	16

Single-table mode

In *single-table* mode, a `single_table_name` configuration key and a `single_table_dimension_name_for_telegraf_measurement_name` configuration key are required.

For this example, assume that:

- `single_table_name : my_readings`
- `single_table_dimension_name_for_telegraf_measurement_name : namespace`

In this case, the following mapping will occur:

- The data will be ingested to a single table with the same name as the value of `single_table_name` configuration key.
- `measure_name` will be stored as a dimension with a name equal to the value of the `single_table_dimension_name_for_telegraf_measurement_name` configuration key
- `location` and `season` will be represented as dimensions
- `temperature`, `humidity`, `no2`, and `pm25` will be represented as measurement names.

The resulting table:

`my_readings`

time	location	season	namespace	measure_name	measure_value::bigint
5:43 pm	us-midwest	summer	weather	temperature	82
5:43 pm	us-midwest	summer	weather	humidity	71
5:43 pm	us-west	NULL	airquality	no2	5
5:43 pm	us-west	NULL	airquality	pm25	16

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, follow the steps below:

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 Telegraph with the Timestreams 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. 16), CLI, or SDKs (p. 20).
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. 16), CLI, or SDKs (p. 20).
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:

```
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:

```
./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.

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

Follow the steps below to configure the JDBC driver.

Topics

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. 24).
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

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 `PreparedStatements`, but this will incur an additional cost with Timestream when retrieving the metadata.	FALSE
Region	The database's region.	us-east-1

SDK Option

The following table describes the SDK Option.

Option	Description	Default
RequestTimeout	The time in milliseconds the AWS SDK will wait for a query 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 `PropertiesFileCredentialsProvider` or `InstanceProfileCredentialsProvider` to use for authentication.	NONE
CustomCredentialsFilePath	The path to a properties file containing AWS security credentials `accessKey` and `secretKey`. This is only required if	NONE

Option	Description	Default
	`AwsCredentialsProviderClass` is specified as `PropertiesFileCredentialsProvider`.	

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 `entityID` field provided in the application metadata. Consider the following example: `entityID = http://www.okta.com//IdpAppID`	NONE
RoleARN	The Amazon Resource Name (ARN) of the role that the caller is assuming.	NONE
IdpARN	The Amazon Resource Name (ARN) of the SAML provider in IAM that describes the Idp.	NONE

SAML-based authentication options for Azure AD

The following table describes the available SAML-based authentication options for Azure AD.

Option	Description	Default
IdpName	The Identity Provider (Idp) name to use for SAML-based authentication. One of `Okta` or `AzureAD`.	NONE

Option	Description	Default
IdpHost	The host name of the specified Idp.	NONE
IdpUserName	The user name for the specified Idp account.	NONE
IdpPassword	The password for the specified Idp account.	NONE
AADApplicationID	The unique id of the 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. 149), 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>;Regio
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 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 Microsoft Azure AD. To use Timestream JDBC single sign-on authentication with Microsoft Azure AD, complete each of the sections listed below.

Topics

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. 100).
- An Okta account (Go to https://www.okta.com/login/ to create an account).
- Access to Amazon Timestream (p. 12).

Now that you have completed the Prerequisites, you may proceed to AWS Account Federation in Okta.

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.

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

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. 100).
- 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. 12).

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**
19. Choose **Grant admin consent for Default Directory**

Setting up IAM Identity Provider and Roles in AWS

Complete each section below to set up IAM for Timestream JDBC single sign-on authentication with Microsoft Azure AD:

Topics

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. 128).

Query Language Reference

Note
This Documentation includes the following third-party documentation from the Trino Software Foundation (formerly Presto Software Foundation) – https://trino.io/foundation.html8 (formerly https://prestosql.io/foundation.html). Licensed under the Apache License, Version 2.0 (the "License"); you may not may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0

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. 193) section.

Topics

Supported data types

Timestream's query language supports the following data types:

Data Type	Description
`int`	Represents a 32-bit integer.
`bigint`	Represent a 64-bit integer.
`boolean`	One of the two truth values of logic, `True` and `False`.

Data Type	Description
`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[T,...]`	Contains one or more elements of a specified data type *T*, where *T* can be any of the data types supported in Timestream.
`row(T,...)`	Contains one or more named fields of data type *T*. The fields may be of any data type supported by Timestream, and are accessed with the dot field reference operator: `.`
`date`	Represents a date in the form *YYYY*-*MM*-*DD*. where *YYYY* is the year, *MM* is the month, and *DD* is the day, respectively. The supported range is from *1970*-*01*-*01* to *2262*-*04*-*11*. *Example:* *1971*-*02*-*03*
`time`	Represents the time of day in UTC. The `time` datatype is represented in the form *HH*.*MM*.*SS*.*sssssssss*. Supports nanosecond precision. *Example:* *17*:*02*:*07*.*496000000*
`timestamp`	Represents an instance in time using nanosecond precision time in UTC, tracking the time since Unix time.

Data Type	Description
`interval`	Represents an interval of time as a string literal *`Xt`*, composed of two parts, *`X`* and *`t`*. *`X`* is an numeric value greater than or equal to `0`, and *`t`* is a unit of time like seconds or hours. The unit of time *`t`* is must be one of the following string literals: • `nanoseconds` • `microseconds` • `milliseconds` • `seconds` • `minutes` • `hours` • `days` • `ns` (same as `nanoseconds`) • `us` (same as `microseconds`) • `ms` (same as `milliseconds`) • `s` (same as `seconds`) • `m` (same as `minutes`) • `h` (same as `hours`) • `d` (same as `days`) *Examples:* *`17s`* *`12seconds`* *`21hours`* *`2d`*
`timeseries[row(timestamp, T,...)]`	Represents the values of a measure recorded over a time interval as an `array` composed of `row` objects. Each `row` contains a `timestamp` and one or more measure values of data type *`T`*, where *`T`* can be any one of `bigint`, `boolean`, `double`, or `varchar`. Rows are assorted in ascending order by `timestamp`. The *timeseries* datatype represents the values of a measure over time.
`unknown`	Represents null data.

Built-in time series functionality

Timestream 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

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

Topics

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_nam	measure_value::double
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	35.0
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	38.2
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	cpu_utilization	45.3
2019-12-04 19:00:00.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	cpu_utilization	54.1
2019-12-04 19:00:01.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	cpu_utilization	42.5
2019-12-04 19:00:02.000000000	us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef1	cpu_utilization	33.7

Running the query:

```
SELECT region, az, vpc, instance_id, CREATE_TIME_SERIES(time, measure_value::double) as
 cpu_utilization FROM metrics
    WHERE measure_name='cpu_utilization'
    GROUP BY region, az, vpc, instance_id
```

will return all series that have `cpu_utilization` as a measure value. In this case, we have two series:

region	az	vpc	instance_id	cpu_utilization
us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	[{time: 2019-12-04 19:00:00.000000000, measure_value::double: 35.0}, {time: 2019-12-04 19:00:01.000000000, measure_value::double: 38.2}, {time: 2019-12-04 19:00:02.000000000, measure_value::double: 45.3}]
us-east-1	us-east-1d	vpc-1a2b3c4d	i-1234567890abcdef0	[{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

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
`interpolate_linear(timeseries, array[timestamp])`	timeseries	Fills in missing data using linear interpolation.
`interpolate_linear(timeseries, timestamp)`	double	Fills in missing data using linear interpolation.

Function	Output data type	Description
`interpolate_spline_cubic(timeseries, array[timestamp])`	timeseries	Fills in missing data using cubic spline interpolation.
`interpolate_spline_cubic(timeseries, timestamp)`	double	Fills in missing data using cubic spline interpolation.
`interpolate_locf(timeseries, array[timestamp])`	timeseries	Fills in missing data using the last sampled value.
`interpolate_locf(timeseries, timestamp)`	double	Fills in missing data using the last sampled value.
`interpolate_fill(timeseries, array[timestamp], double)`	timeseries	Fills in missing data using a constant value.
`interpolate_fill(timeseries, timestamp, double)`	double	Fills in missing data using a constant value.

Query examples

Example

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using linear interpolation:

```
WITH binned_timeseries AS (
SELECT hostname, BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double), 2)
 AS avg_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
    AND time > ago(2h)
GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
SELECT hostname,
    INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
            SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
FROM binned_timeseries
GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Example

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using interpolation based on the last observation carried forward:

```
WITH binned_timeseries AS (
SELECT hostname, BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double), 2)
 AS avg_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
```

```
    AND time > ago(2h)
GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
SELECT hostname,
    INTERPOLATE_LOCF(
        CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
            SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
FROM binned_timeseries
GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Derivatives functions

Derivatives are used calculate the rate of change for a given metric and can be used to proactively respond to an event. For example, suppose you calculate the derivative of the CPU utilization of EC2 instances over the past 5 minutes, and you notice a significant positive derivative. This can be indicative of increased demand on your workload, so you may decide want to spin up more EC2 instances to better handle your workload.

Amazon Timestream supports two variants of derivative functions. This section provides usage information for the Timestream derivative functions, as well as sample queries.

Usage information

Function	Output data type	Description
`derivative_linear(timeseries, interval)`	timeseries	Calculates the derivative of each point in the `timeseries` for the specified `interval`.
`non_negative_derivative_linear(timeseries, interval)`	timeseries	Same as `derivative_linear(timeseries, interval)`, but only returns positive values.

Query examples

Example

Find the rate of change in the CPU utilization every 5 minutes over the past 1 hour:

```
SELECT DERIVATIVE_LINEAR(CREATE_TIME_SERIES(time, measure_value::double), 5m) AS result
FROM “sampleDB”.DevOps
WHERE measure_name = 'cpu_utilization'
AND hostname = 'host-Hovjv' and time > ago(1h)
GROUP BY hostname, measure_name
```

Example

Calculate the rate of increase in errors generated by one or more microservices:

```
WITH binned_view as (
    SELECT bin(time, 5m) as binned_timestamp, ROUND(AVG(measure_value::double), 2) as value
```

```
    FROM “sampleDB”.DevOps
    WHERE micro_service = 'jwt'
    AND time > ago(1h)
    AND measure_name = 'service_error'
    GROUP BY bin(time, 5m)
)
SELECT non_negative_derivative_linear(CREATE_TIME_SERIES(binned_timestamp, value), 1m) as
 rateOfErrorIncrease
FROM binned_view
```

Integral functions

You can use integrals to find the area under the curve per unit of time for your time series events. As an example, suppose you're tracking the volume of requests received by your application per unit of time. In this scenario, you can use the integral function to determine the total volume of requests served 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
`integral_trapezoidal(timeseries, interval)`	double	Approximates the integral over the specified `interval` for the `timeseries` 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
`correlate_pearson(timeseries, timeseries)`	double	Calculates Pearson's correlation coefficient for the two `timeseries`. The timeseries must have the same timestamps.
`correlate_spearman(timeseries, timeseries)`	double	Calculates Spearman's correlation coefficient for the two `timeseries`. The timeseries must have the same timestamps.

Query examples

Example

```
WITH cte_1 AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, measure_value::double),
        SEQUENCE(min(time), max(time), 10m)) AS result
    FROM sample.DevOps
    WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv' AND time > ago(1h)
    GROUP BY hostname, measure_name
),
cte_2 AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, measure_value::double),
        SEQUENCE(min(time), max(time), 10m)) AS result
    FROM sample.DevOps
    WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv' AND time > ago(1h)
    GROUP BY hostname, measure_name
)
SELECT correlate_pearson(cte_1.result, cte_2.result) AS result
FROM cte_1, cte_2
```

Filter and reduce functions

Amazon Timestream supports functions for performing filter and reduce operations on time series data. This section provides usage information for the Timestream filter and reduce functions, as well as sample queries.

Usage information

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

Function	Output data type	Description
`reduce(timeseries(T), initialState S, inputFunction(S, T, S), outputFunction(S, R))`	R	Returns a single value, reduced from the time series. The `inputFunction` will be invoked on each element in timeseries in order. In addition to taking the current element, inputFunction takes the current state (initially `initialState`) and returns the new state. The `outputFunction` will be invoked to turn the final state into the result value. The `outputFunction` can be an identity function.

Query examples

Example

Construct a time series of CPU utilization of a host and filter points with measurement greater than 70:

```
WITH time_series_view AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, ROUND(measure_value::double,2)),
            SEQUENCE(ago(15m), ago(1m), 10s)) AS cpu_user
    FROM sample.DevOps
    WHERE hostname = 'host-Hovjv' and measure_name = 'cpu_utilization'
        AND time > ago(30m)
    GROUP BY hostname
)
SELECT FILTER(cpu_user, x -> x.value > 70.0) AS cpu_above_threshold
from time_series_view
```

Example

Construct a time series of CPU utilization of a host and determine the sum squared of the measurements:

```
WITH time_series_view AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, ROUND(measure_value::double,2)),
            SEQUENCE(ago(15m), ago(1m), 10s)) AS cpu_user
    FROM sample.DevOps
    WHERE hostname = 'host-Hovjv' and measure_name = 'cpu_utilization'
        AND time > ago(30m)
    GROUP BY hostname
)
SELECT REDUCE(cpu_user,
    DOUBLE '0.0',
    (s, x) -> x.value * x.value + s,
    s -> s)
from time_series_view
```

Example

Construct a time series of CPU utilization of a host and determine the fraction of samples that are above the CPU threshold:

```
WITH time_series_view AS (
    SELECT INTERPOLATE_LINEAR(
        CREATE_TIME_SERIES(time, ROUND(measure_value::double,2)),
            SEQUENCE(ago(15m), ago(1m), 10s)) AS cpu_user
    FROM sample.DevOps
    WHERE hostname = 'host-Hovjv' and measure_name = 'cpu_utilization'
        AND time > ago(30m)
    GROUP BY hostname
)
SELECT ROUND(
    REDUCE(cpu_user,
      -- initial state
      CAST(ROW(0, 0) AS ROW(count_high BIGINT, count_total BIGINT)),
      -- function to count the total points and points above a certain threshold
      (s, x) -> CAST(ROW(s.count_high + IF(x.value > 70.0, 1, 0), s.count_total + 1) AS
 ROW(count_high BIGINT, count_total BIGINT)),
      -- output function converting the counts to fraction above threshold
      s -> IF(s.count_total = 0, NULL, CAST(s.count_high AS DOUBLE) / s.count_total)),
    4) AS fraction_cpu_above_threshold
from time_series_view
```

SQL support

Timestream supports common SQL constructs. You can read more below:

Topics

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. 191).
- `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
null	false	false	null
true	null	null	true

A	B	A and B	A or B
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()

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

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.

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()

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.
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.

Function	Output data type	Description
round(x,d)	[same as input]	Returns x rounded to d decimal places.
sign(x)	[same as input]	Returns the signum function of x, that is: • 0 if the argument is 0 • 1 if the argument is greater than 0 • -1 if the argument is less than 0. For double arguments, the function additionally returns: • NaN if the argument is NaN • 1 if the argument is +Infinity • -1 if the argument is -Infinity.
sqrt(x)	double	Returns the square root of x.
to_base(x, radix)	varchar	Returns the base-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.
sin(x)	double	Returns the sine of x.
tan(x)	double	Returns the tangent of x.
tanh(x)	double	Returns the hyperbolic tangent of x.
infinity()	double	Returns the constant representing positive infinity.
is_finite(x)	boolean	Determine if x is finite.
is_infinite(x)	boolean	Determine if x is infinite.
is_nan(x)	boolean	Determine if x is not-a-number.

Function	Output data type	Description
nan()	double	Returns the constant representing not-a-number.

String Operators

Timestream supports the || operator for concatenating one or strings.

String Functions

Note
The input data type of these functions is assumed to be varchar unless otherwise specified.

Function	Output data type	Description
chr(n)	varchar	Returns the Unicode code point n as a varchar.
codepoint(x)	integer	Returns the Unicode code point of the only character of str.
concat(x1, ..., xN)	varchar	Returns the concatenation of x1, x2, ..., xN.
hamming_distance(x1,x2)	bigint	Returns the Hamming distance of x1 and x2, i.e. the number of positions at which the corresponding characters are different. Note that the two varchar inputs must have the same length.
length(x)	bigint	Returns the length of x in characters.
levenshtein_distance(x1, x2)	bigint	Returns the Levenshtein edit distance of x1 and x2, i.e. the minimum number of single-character edits (insertions, deletions or substitutions) needed to change x1 into x2.
lower(x)	varchar	Converts x to lowercase.
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.

Function	Output data type	Description
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.
strpos(x1, x2,bigint instance)	bigint	Returns the position of the Nth instance of x2 in x1. Instance must be a positive number. Positions start with 1. If not found, 0 is returned.
strrpos(x1, x2)	bigint	Returns the starting position of the last instance of x2 in x1. Positions start with 1. If not found, 0 is returned.
strrpos(x1, x2, bigint instance)	bigint	Returns the position of the Nth instance of x2 in x1 starting from the end of x1. instance must be a positive number. Positions start with 1. If not found, 0 is returned.

Function	Output data type	Description
position(x2 IN x1)	bigint	Returns the starting position of the first instance of x2 in x1. Positions start with 1. If not found, 0 is returned.
substr(x, bigint start)	varchar	Returns the rest of x from the starting position start. Positions start with 1. A negative starting position is interpreted as being relative to the end of x.
substr(x, bigint start, bigint len)	varchar	Returns a substring from x of length len from the starting position start. Positions start with 1. A negative starting position is interpreted as being relative to the end of x.
trim(x)	varchar	Removes leading and trailing whitespace from x.
upper(x)	varchar	Converts x to uppercase.

Array Operators

Timestream 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.

Function	Output data type	Description
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, 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.

Function	Output data type	Description
sequence(start, stop)	array(timestamp)	Generate a sequence of timestamps from start date to stop date, incrementing by 1 day.
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.

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 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

Function	Output data type	Description
		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).
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

Timestream supports the `timestamp` and `interval` data types for date/time data types.

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

Timestream supports the following operations on `timestamps` and `intervals`:

Operator	Description
+	Addition
-	Subtraction

Date / Time Functions

Timestream only supports the timestamp and interval data types for date/time data types.

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

Timestream supports the following functions on `timestamps` and `intervals`:

Function	Output data type	Description
current_date	timestamp	Returns Timestamp for UTC 00:00:00 for the current day.
current_timestamp or now()	timestamp	Returns Timestamp for current time in UTC.
from_iso8601_timestamp(string)	timestamp	Parses the ISO 8601 timestamp into internal timestamp format.
from_iso8601_date(string)	timestamp	Parses the ISO 8601 date string into internal Timestamp format for UTC 00:00:00 of the specified date.
to_iso8601(x)	varchar	Returns an ISO 8601 formatted string for the input timestamp x.
date_trunc(unit, timestamp)	timestamp	Returns the timestamp truncated to unit, where unit is one of [second, minute, hour, day, week, month, quarter, or year].
parse_duration(string)	interval	Parses the input string to return an `interval` equivalent.
bin(timestamp, interval)	timestamp	Rounds value down to a multiple of the given bin `interval`.
ago(interval)	timestamp	Returns the value corresponding to current_timestamp `interval`.
interval literals such as 1h, 1d, 30m, etc.	interval	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.

Function	Output data type	Description
		For example, `ago(1d)` and `bin(time, 1m)`.

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	count(*) returns the number of input rows.count(x) returns the number of non-null input values.
count_if(x)	bigint	Returns the number of TRUE input values.
geometric_mean(x)	double	Returns the geometric mean of all input values.
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.

Function	Output data type	Description
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.
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].

Function	Output data type	Description
approx_percentile(x, percentage)	[same as x]	Returns the approximate percentile for all input values of x at the given percentage. The value of percentage must be between zero and one and must be constant for all input rows.
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.
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.

Function	Output data type	Description
approx_percentile(x, w, percentage, accuracy)	[same as x]	Returns the approximate weighed percentile for all input values of x using the per-item weight w at the percentage p, with a maximum rank error of accuracy. The weight must be an integer value of at least one. It is effectively a replication count for the value x in the percentile set. The value of p must be between zero and one and must be constant for all input rows. 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.
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 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

Function	Output data type	Description
		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.
last_value(x)*	[same as input]	Returns the last value of the window.
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.
lead(x[, offset[, default_value]])	[same as input]	Returns the value at offset rows after the current row in the window. Offsets start at 0, which is the current row. The offset can be any scalar expression. The default offset is 1. If the offset is null or larger than the window, the default_value is returned, or if it is not specified null is returned.
lag(x[, offset[, default_value]])	[same as input]	Returns the value at offset rows before the current row in the window Offsets start at 0, which is the current row. The offset can be any scalar expression. The default offset is 1. If the offset is null or larger than the window, the default_value is returned, or if it is not specified null is returned.

Sample Queries

This section includes example use cases of Timestream's Query Language.

Topics

Simple queries

To get the 10 most recently added data points for a table:

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

To get 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
```

To manipulate 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
```

Queries with time series functions

Topics

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_nam	measure_valu	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	35.1	null

time	region	az	hostname	measure_nam	measure_valu	measure_value::bigint
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	memory_utilization	85.3	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	network_bytes_in	null	1,500
2019-12-04 19:00:00.000000000	us-east-1	us-east-1a	frontend01	network_bytes_out	null	6,700
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	cpu_utilization	38.5	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	memory_utilization	88.4	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	network_bytes_in	null	23,000
2019-12-04 19:00:00.000000000	us-east-1	us-east-1b	frontend02	network_bytes_out	null	12,000
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	cpu_utilization	45.0	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	memory_utilization	85.8	null
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	network_bytes_in	null	15,000
2019-12-04 19:00:00.000000000	us-east-1	us-east-1c	frontend03	network_bytes_out	null	836,000
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	55.2	null
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	memory_utilization	75.0	null
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	network_bytes_in	null	1,245
2019-12-04 19:00:05.000000000	us-east-1	us-east-1a	frontend01	network_bytes_out	null	68,432
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	cpu_utilization	65.6	null
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	memory_utilization	85.3	null
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	network_bytes_in	null	1,245
2019-12-04 19:00:08.000000000	us-east-1	us-east-1b	frontend02	network_bytes_out	null	68,432
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	cpu_utilization	12.1	null

time	region	az	hostname	measure_nam	measure_valu	measure_value::bigint
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	memory_utilization	32.0	null
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	network_bytes_in	null	1,400
2019-12-04 19:00:20.000000000	us-east-1	us-east-1c	frontend03	network_bytes_out	null	345
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	cpu_utilization	15.3	null
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	memory_utilization	35.4	null
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	network_bytes_in	null	23
2019-12-04 19:00:10.000000000	us-east-1	us-east-1a	frontend01	network_bytes_out	null	0
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	cpu_utilization	44.0	null
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	memory_utilization	64.2	null
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	network_bytes_in	null	1,450
2019-12-04 19:00:16.000000000	us-east-1	us-east-1b	frontend02	network_bytes_out	null	200
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	cpu_utilization	66.4	null
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	memory_utilization	86.3	null
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	network_bytes_in	null	300
2019-12-04 19:00:40.000000000	us-east-1	us-east-1c	frontend03	network_bytes_out	null	423

Find the average, p90, p95, and p99 CPU utilization for a specific EC2 host over the past 2 hours:

```
SELECT region, az, hostname, BIN(time, 15s) AS binned_timestamp,
    ROUND(AVG(measure_value::double), 2) AS avg_cpu_utilization,
    ROUND(APPROX_PERCENTILE(measure_value::double, 0.9), 2) AS p90_cpu_utilization,
    ROUND(APPROX_PERCENTILE(measure_value::double, 0.95), 2) AS p95_cpu_utilization,
    ROUND(APPROX_PERCENTILE(measure_value::double, 0.99), 2) AS p99_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
    AND time > ago(2h)
GROUP BY region, hostname, az, BIN(time, 15s)
ORDER BY binned_timestamp ASC
```

Identify EC2 hosts with CPU utilization that is higher by 10 % or more compared to the average CPU utilization of the entire fleet for the past 2 hours:

```
WITH avg_fleet_utilization AS (
    SELECT COUNT(DISTINCT hostname) AS total_host_count, AVG(measure_value::double) AS
 fleet_avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND time > ago(2h)
), avg_per_host_cpu AS (
    SELECT region, az, hostname, AVG(measure_value::double) AS avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND time > ago(2h)
    GROUP BY region, az, hostname
)
SELECT region, az, hostname, avg_cpu_utilization, fleet_avg_cpu_utilization
FROM avg_fleet_utilization, avg_per_host_cpu
WHERE avg_cpu_utilization > 1.1 * fleet_avg_cpu_utilization
ORDER BY avg_cpu_utilization DESC
```

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours:

```
SELECT BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double), 2) AS
 avg_cpu_utilization
FROM "sampleDB".DevOps
WHERE measure_name = 'cpu_utilization'
    AND hostname = 'host-Hovjv'
    AND time > ago(2h)
GROUP BY hostname, BIN(time, 30s)
ORDER BY binned_timestamp ASC
```

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using linear interpolation:

```
WITH binned_timeseries AS (
    SELECT hostname, BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double),
 2) AS avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND hostname = 'host-Hovjv'
        AND time > ago(2h)
    GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
    SELECT hostname,
        INTERPOLATE_LINEAR(
            CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
                SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
    FROM binned_timeseries
    GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Find the average CPU utilization binned at 30 second intervals for a specific EC2 host over the past 2 hours, filling in the missing values using interpolation based on the last observation carried forward:

```
WITH binned_timeseries AS (
```

```
    SELECT hostname, BIN(time, 30s) AS binned_timestamp, ROUND(AVG(measure_value::double),
 2) AS avg_cpu_utilization
    FROM "sampleDB".DevOps
    WHERE measure_name = 'cpu_utilization'
        AND hostname = 'host-Hovjv'
        AND time > ago(2h)
    GROUP BY hostname, BIN(time, 30s)
), interpolated_timeseries AS (
    SELECT hostname,
        INTERPOLATE_LOCF(
            CREATE_TIME_SERIES(binned_timestamp, avg_cpu_utilization),
                SEQUENCE(min(binned_timestamp), max(binned_timestamp), 15s)) AS
 interpolated_avg_cpu_utilization
    FROM binned_timeseries
    GROUP BY hostname
)
SELECT time, ROUND(value, 2) AS interpolated_cpu
FROM interpolated_timeseries
CROSS JOIN UNNEST(interpolated_avg_cpu_utilization)
```

Queries with aggregate functions

Below is an example IoT scenario example data set to illustrate queries with aggregate functions.

Topics

Example Data

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::varch
2019-12-04 19:00:00.000000000	1	123456781	GMC	Astro	Alpha	100	500	fuel_reading	65.2	null
2019-12-04 19:00:00.000000000	1	123456781	GMC	Astro	Alpha	100	500	load	400.0	null
2019-12-04 19:00:00.000000000	1	123456781	GMC	Astro	Alpha	100	500	speed	90.2	null
2019-12-04 19:00:00.000000000	1	123456781	GMC	Astro	Alpha	100	500	location	null	47.6062 N, 122.3321 W
2019-12-04 19:00:00.000000000	2	123456782	Kenworth	W900	Alpha	150	1000	fuel_reading	10.1	null
2019-12-04 19:00:00.000000000	2	123456782	Kenworth	W900	Alpha	150	1000	load	950.3	null

time	truck_id	make	model	fleet	fuel_capa	load_cap	measure_	measure_	measure_value::varch
2019-12-04 19:00:00.000000000	123456788	Kenworth	W900	Alpha	150	1000	speed	50.8	null
2019-12-04 19:00:00.000000000	123456788	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
```

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

- Amazon Timestream is designed to collect, store, and analyze time series data from applications and devices emitting a sequence of data with a timestamp that changes over time. For optimal performance, the data being sent to Timestream must have temporal characteristics and time must be a quintessential component of the data analysis requirements.
- When deciding which attributes of your time series data map to measures and dimensions, consider the following:
  - Represent all metadata attributes as dimensions. Examples are the region and availability zone of an EC2 instance, the stock exchange name, the device ID and make of an IoT sensor.
  - Represent all measurements as measures. Examples are the CPU and memory utilization of an EC2 instance, the stock price, and the temperature/humidity reading of an IoT sensor.
  - Low cardinality attributes that are not measurements and do not contain metadata can be represented as dimensions. Examples include measurement quality (high, low, medium) or stock purchasing recommendation (buy, sell, hold). In this case, however, your time series data will consist of 3 time series that will need to be combined together using the UNION clause while running queries.
- When deciding whether to create a single table or multiple tables to store data consider the following:
  - Consider the access control requirements of your application. Data that requires to be encrypted using different AWS KMS keys must be placed in separate databases.
  - Consider the data retention requirements of your application. Data that requires different retention policies must be place in different tables.
  - Unrelated data must be stored in separate tables.
  - Data that is queried together must be stored in the same table.
- Keep the dimension names shorter to save on data ingestion and storage costs

- Store Boolean measure values (a 0 or 1 state) using the Boolean data type, rather than the bigint data type. This optimizes your Timestream application for cost.
- When ingesting data into Timestream, note that a single measure name can only be associated with one type of measure value. For example, if you have a measure value named *cpu_user* that is ingested into Timestream as a double, the measure value must remain a double, and cannot be converted to another data type.
- Whenever possible, consider modeling your Timestream data using many measures per table whenever possible. This allows for better pruning.

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. 131)

Configuring 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.
  - 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 later than data retention configured for the memory store and no earlier than the future ingestion period defined in Quotas (p. 211). Sending data with a timestamp outside these bounds will result in the data being rejected by Timestream
- 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. 211).
  - 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 throttling. If your application experiences this 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

Queries

- 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
- 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
- Use a CASE expression to perform complex aggregations instead of selecting from the same table multiple times
- 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)
- 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
- 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. Refer to the Query API documentation for more information
- 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
- 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

Client applications and supported integrations

- Run your client application from the same region as Timestream to reduce network latencies and data transfer costs
- 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

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

Consider a time series event representing the CPU utilization of an EC2 instance as shown below:

time	region	az	vpc	hostname	measure_nam	measure_value::double
1602983435238563001	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. 205), 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

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 = 6,220,800 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. 205)), 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.
- 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

Timestream Specific Error Codes

This section contains the specific error codes for Timestream.

Timestream Write API Errors

InternalServerException

HTTP Status Code: 500

ThrottlingException

HTTP Status Code: 429

ValidationException

HTTP Status Code: 400

ConflictException

HTTP Status Code: 409

AccessDeniedException

You do not have sufficient access to perform this action.

HTTP Status Code: 403

ServiceQuotaExceededException

HTTP Status Code: 402

ResourceNotFoundException

HTTP Status Code: 404

RejectedRecordsException

HTTP Status Code: 419

InvalidEndpointException

HTTP Status Code: 421

Timestream Query API Errors

ValidationException

HTTP Status Code: 400

QueryExecutionException

HTTP Status Code: 400

ConflictException

HTTP Status Code: 409

ThrottlingException

HTTP Status Code: 429

InternalServerException

HTTP Status Code: 500

InvalidEndpointException

HTTP Status Code: 421

Quotas

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

Topics

Default Quotas

The following table contains the Timestream quotas and the default values.

displayName	description	defaultValue
Databases per account	The maximum number of databases you can create per AWS account.	500
Tables per account	The maximum number of tables you can create per AWS account.	50000
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.	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
Measures per table	The maximum number of measures 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
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 API requests allowed per second per account, in the current region.	1
Dimension name length	The maximum number of characters for a Dimension name.	60 Bytes
Measure name length	The maximum number of characters for a Measure name.	256 Bytes
Database name length	The maximum number of characters for a Database name.	256 Bytes
Table name length	The maximum number of characters 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

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
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	• Must not contain System identifiers (p. 215). • Can contain a-z A-Z 0-9 _ (underscore) - (dash) . (dot). • All names must be encoded as UTF-8, and are case sensitive. **Note** Table and database names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.
Measure Name	• Must not contain System identifiers (p. 215) or colon ':'. • Must not start with a reserved prefix (`ts_`, `measure_value`). **Note** Table and database names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.
Dimension Name	• Must not contain System identifiers (p. 215), 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".

Description	Timestream value
	Note Dimension and measure names are compared using UTF-8 binary representation. This means that comparison for ASCII characters is case sensitive.

Reserved keywords

All of the following are reserved keywords:

- ALTER
- AND
- AS
- BETWEEN
- BY
- CASE
- CAST
- CONSTRAINT
- CREATE
- CROSS
- CUBE
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- DEALLOCATE
- DELETE
- DESCRIBE
- DISTINCT
- DROP
- ELSE
- END
- ESCAPE
- EXCEPT
- EXECUTE
- EXISTS
- EXTRACT
- FALSE
- FOR
- FROM
- FULL
- GROUP
- GROUPING
- HAVING
- IN

- INNER
- INSERT
- INTERSECT
- INTO
- IS
- JOIN
- LEFT
- LIKE
- LOCALTIME
- LOCALTIMESTAMP
- NATURAL
- NORMALIZE
- NOT
- NULL
- ON
- OR
- ORDER
- OUTER
- PREPARE
- RECURSIVE
- RIGHT
- ROLLUP
- SELECT
- TABLE
- THEN
- TRUE
- UESCAPE
- UNION
- UNNEST
- USING
- VALUES
- WHEN
- WHERE
- WITH

System identifiers

We reserve column names “measure_value”, "ts_non_existent_col" and “time” to be Timestream 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 database, table, dimension or measure names.

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. 209) For error codes common to all AWS services, see the AWS Support section.

Topics

Actions

The following actions are supported by Amazon Timestream Write:

The following actions are supported by Amazon Timestream Query:

- CancelQuery (p. 257)
- DescribeEndpoints (p. 259)
- Query (p. 261)

Amazon Timestream Write

The following actions are supported by Amazon Timestream Write:

- CreateDatabase (p. 218)
- CreateTable (p. 221)
- DeleteDatabase (p. 224)
- DeleteTable (p. 226)
- DescribeDatabase (p. 228)
- DescribeEndpoints (p. 230)
- DescribeTable (p. 232)
- ListDatabases (p. 235)
- ListTables (p. 238)
- ListTagsForResource (p. 241)
- TagResource (p. 243)
- UntagResource (p. 245)
- UpdateDatabase (p. 247)
- UpdateTable (p. 250)
- WriteRecords (p. 253)

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. Refer to AWS managed AWS KMS keys for more info. Service quotas apply. See code sample for details.

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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 218)

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. 218)

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. Refer to AWS managed AWS KMS keys for more info.

Type: String

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

Required: No

Tags (p. 218)

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

Type: Array of Tag (p. 278) 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. 219)

The newly created Timestream database.

Type: Database (p. 267) object

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

HTTP Status Code: 400

InvalidEndpointException

The requested endpoint was invalid.

HTTP Status Code: 400

ServiceQuotaExceededException

Instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user exceeding service quotas. The request was throttled.

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

CreateTable

Service: Amazon Timestream Write

The CreateTable operation 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 may 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",
   "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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 221)

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

RetentionProperties (p. 221)

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

Type: RetentionProperties (p. 275) object

Required: No

TableName (p. 221)

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. 221)

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

Type: Array of Tag (p. 278) 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,
      "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. 222)

The newly created Timestream table.

Type: Table (p. 276) object

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

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

Instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user exceeding service quotas. The request was throttled.

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

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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 224)

The name of the Timestream database to be deleted.

Type: String

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

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

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. 285).

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 invalid.

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 exceeding service quotas. The request was throttled.

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

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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 226)

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.

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

Required: Yes

TableName (p. 226)

The name of the Timestream table to be deleted.

Type: String

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

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

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. 285).

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 invalid.

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 exceeding service quotas. The request was throttled.

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

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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 228)

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

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. 228)

The name of the Timestream table.

Type: Database (p. 267) object

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

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 exceeding service quotas. The request was throttled.

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 Write

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. 230)

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

Type: Array of Endpoint (p. 270) objects

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

InternalServerException

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 exceeding service quotas. The request was throttled.

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

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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 232)

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

TableName (p. 232)

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

Response Syntax

```
{
   "Table": {
      "Arn": "string",
      "CreationTime": number,
      "DatabaseName": "string",
      "LastUpdatedTime": number,
      "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. 232)

The Timestream table.

Type: Table (p. 276) object

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

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 exceeding service quotas. The request was throttled.

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

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. 287).

The request accepts the following data in JSON format.

MaxResults (p. 235)

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. 235)

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. 235)

A list of database names.

Type: Array of Database (p. 267) objects

NextToken (p. 235)

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. 285).

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 invalid.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user exceeding service quotas. The request was throttled.

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

ListTables

Service: Amazon Timestream Write

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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 238)

The name of the Timestream database.

Type: String

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

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

Required: No

MaxResults (p. 238)

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. 238)

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,
         "DatabaseName": "string",
         "LastUpdatedTime": number,
         "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. 238)

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

Type: String

Tables (p. 238)

A list of tables.

Type: Array of Table (p. 276) objects

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

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 exceeding service quotas. The request was throttled.

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 Write

List 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. 287).

The request accepts the following data in JSON format.

ResourceARN (p. 241)

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. 241)

The tags currently associated with the Timestream resource.

Type: Array of Tag (p. 278) 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. 285).

InvalidEndpointException

The requested endpoint was invalid.

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 exceeding service quotas. The request was throttled.

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 Write

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. 287).

The request accepts the following data in JSON format.

ResourceARN (p. 243)

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. 243)

The tags to be assigned to the Timestream resource.

Type: Array of Tag (p. 278) 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. 285).

InvalidEndpointException

The requested endpoint was invalid.

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

Instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user exceeding service quotas. The request was throttled.

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 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. 287).

The request accepts the following data in JSON format.

ResourceARN (p. 245)

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. 245)

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. 285).

InvalidEndpointException

The requested endpoint was invalid.

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

Instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user exceeding service quotas. The request was throttled.

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

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. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 247)

The name of the database.

Type: String

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

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

Required: Yes

KmsKeyId (p. 247)

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/ExampleAlias`

Type: String

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

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. 247)

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. 267) object

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

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

Instance quota of resource exceeded for this account.

HTTP Status Code: 400

ThrottlingException

Too many requests were made by a user exceeding service quotas. The request was throttled.

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

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

Request Parameters

For information about the parameters that are common to all actions, see Common Parameters (p. 287).

The request accepts the following data in JSON format.

DatabaseName (p. 250)

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

RetentionProperties (p. 250)

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

Type: RetentionProperties (p. 275) object

Required: Yes

TableName (p. 250)

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

Response Syntax

```
{
   "Table": {
      "Arn": "string",
      "CreationTime": number,
      "DatabaseName": "string",
      "LastUpdatedTime": number,
      "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. 251)

The updated Timestream table.

Type: Table (p. 276) object

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

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 exceeding service quotas. The request was throttled.

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

WriteRecords

Service: Amazon Timestream Write

The WriteRecords operation 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 with 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 support 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 not specified for the record in the request. Timestream will update an existing record's measure value along with its `Version` upon receiving a write request with a higher `Version` number for that record. Upon receiving 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 will store 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",
      "MeasureValueType": "string",
      "Time": "string",
      "TimeUnit": "string",
      "Version": number
   },
   "DatabaseName": "string",
   "Records": [
      {
         "Dimensions": [
            {
               "DimensionValueType": "string",
```

```
                "Name": "string",
                "Value": "string"
            }
        ],
        "MeasureName": "string",
        "MeasureValue": "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. 287).

The request accepts the following data in JSON format.

CommonAttributes (p. 253)

A record containing 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. 271) object

Required: No

DatabaseName (p. 253)

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

Records (p. 253)

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

Type: Array of Record (p. 271) objects

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

Required: Yes

TableName (p. 253)

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

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. 285).

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 invalid.

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 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 exceeding service quotas. The request was throttled.

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

Amazon Timestream Query

The following actions are supported by Amazon Timestream Query:

- CancelQuery (p. 257)
- DescribeEndpoints (p. 259)
- Query (p. 261)

CancelQuery

Service: Amazon Timestream Query

Cancels a query that has been issued. Cancellation is guaranteed only if the query has not completed execution 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. 287).

The request accepts the following data in JSON format.

QueryId (p. 257)

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. 257)

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. 285).

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 invalid.

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. 259)

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

Type: Array of Endpoint (p. 281) objects

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

InternalServerException

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

Query

Service: Amazon Timestream Query

`Query` is a synchronous operation that enables you to execute a query against your Amazon Timestream data. `Query` will timeout 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 1MB, 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. 287).

The request accepts the following data in JSON format.

ClientToken (p. 261)

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*, meaning that executing 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 execution.
- 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. 261)

The total number of rows to be returned in the `Query` output. Initial execution 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 amount of rows to meet the 1 MB limit.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 1000.

Required: No

NextToken (p. 261)

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 execution.

Note the following when using NextToken in a query:

- A pagination token can be used for up to 5 `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 to 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. 261)

The query to be executed 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. 263)

The column data types of the returned result set.

Type: Array of ColumnInfo (p. 279) objects

NextToken (p. 263)

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. 263)

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. 263)

Information about the status of the query, including progress and bytes scanned.

Type: QueryStatus (p. 282) object

Rows (p. 263)

The result set rows returned by the query.

Type: Array of Row (p. 283) objects

Errors

For information about the errors that are common to all actions, see Common Errors (p. 285).

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 invalid.

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

Data Types

The following data types are supported by Amazon Timestream Write:

- Database (p. 267)
- Dimension (p. 269)
- Endpoint (p. 270)
- Record (p. 271)
- RejectedRecord (p. 273)
- RetentionProperties (p. 275)
- Table (p. 276)
- Tag (p. 278)

The following data types are supported by Amazon Timestream Query:

- ColumnInfo (p. 279)
- Datum (p. 280)
- Endpoint (p. 281)
- QueryStatus (p. 282)
- Row (p. 283)
- TimeSeriesDataPoint (p. 284)
- Type (p. 285)

Amazon Timestream Write

The following data types are supported by Amazon Timestream Write:

- Database (p. 267)
- Dimension (p. 269)
- Endpoint (p. 270)
- Record (p. 271)
- RejectedRecord (p. 273)
- RetentionProperties (p. 275)

- Table (p. 276)
- Tag (p. 278)

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.

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

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

Dimension represents the meta data 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 meta data 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

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

Endpoint

Service: Amazon Timestream Write

Represents an available endpoint against which to make API calls agaisnt, 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

Record

Service: Amazon Timestream Write

Record represents a time series data point being written into Timestream. Each record contains an array of dimensions. Dimensions represent the meta data 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. A record also contains the measure value and the value type which is the data type of the measure value. In addition, the record contains the timestamp when the measure was collected that 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. 269) 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

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`

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

RejectedRecord

Service: Amazon Timestream Write

Records that were not successfully inserted into Timestream due to data validation issues that must be resolved prior to 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

Table

Service: Amazon Timestream Write

Table 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.

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

Required: No

LastUpdatedTime

The time when the Timestream table was last updated.

Type: Timestamp

Required: No

RetentionProperties

The retention duration for the memory store and magnetic store.

Type: RetentionProperties (p. 275) object

Required: No

TableName

The name of the Timestream table.

Type: String

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

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

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. 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

Amazon Timestream Query

The following data types are supported by Amazon Timestream Query:

ColumnInfo

Service: Amazon Timestream Query

Contains the meta data 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. 285) 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. 280) 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. 283) 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 of timeseries data type.

Type: Array of TimeSeriesDataPoint (p. 284) 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

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

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 execution 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. 280) 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

TimeSeriesDataPoint

Service: Amazon Timestream Query

The timeseries datatype 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 timeseries. It represents a tuple of (time, measure value) in a timeseries.

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. 280) 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

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

ArrayColumnInfo

Indicates if the column is an array.

Type: ColumnInfo (p. 279) object

Required: No

RowColumnInfo

Indicates if the column is a row.

Type: Array of ColumnInfo (p. 279) 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. 279) 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

ValidationError

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

update-history-change	update-history-description	update-history-date
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