Alibaba Cloud Tablestore

Developer Guide

Issue: 20190919

Legal disclaimer

or trust in this document, even if Alibaba Cloud has been notified of the possibility of such a loss.

5. By law, all the content of the Alibaba Cloud website, including but not limited to works, products, images, archives, information, materials, website architecture, website graphic layout, and webpage design, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of the Alibaba Cloud website, product programs, or content shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates).
6. Please contact Alibaba Cloud directly if you discover any errors in this document.

Generic conventions

Table -1: Style conventions

Style	Description	Example
	This warning information indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	Danger: Resetting will result in the loss of user configuration data.
	This warning information indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	Warning: Restarting will cause business interruption. About 10 minutes are required to restore business.
	This indicates warning information, supplementary instructions, and other content that the user must understand.	Notice: Take the necessary precautions to save exported data containing sensitive information.
	This indicates supplemental instructions, best practices, tips, and other content that is good to know for the user.	Note: You can use Ctrl + A to select all files.
>	Multi-level menu cascade.	Settings > Network > Set network type
Bold	It is used for buttons, menus, page names, and other UI elements.	Click OK.
`Courier font`	It is used for commands.	Run the `cd / d C :/ windows` command to enter the Windows system folder.
Italics	It is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	It indicates that it is a optional value, and only one item can be selected.	`ipconfig` *[-all|-t]*

Style	Description	Example
{} or {a|b}	It indicates that it is a required value, and only one item can be selected.	`swich` *{stand | slave}*

Contents

1 Overview

Table Store is a NoSQL multi-model database service independently developed by Alibaba Cloud. Table Store can store large amounts of structured data and provide query and analysis services. The distributed storage and powerful index-based search engine enable Table Store to store PB-grade data while ensuring 10 million TPS and millisecond-level latency. This document introduces terms, models, and features of Table Store.

Terms

The following table describes the terms for Table Store.

Term	Description
#unique_4	An instance is an entity used to manage tables and data in Table Store. Each instance is equivalent to a database. Table Store implements access control and resource metering for applications at the instance level.
#unique_5	The read/write throughput is measured by read/write capacity units (CUs), which is the smallest billing unit for read and write operations.
#unique_6	A region is a physical data center of Alibaba Cloud.
#unique_7	Each Table Store instance has an endpoint. An endpoint must be specified before any operations can be performed on tables or data in Table Store.

Models

Table Store provides multiple models that you can apply for as needed. The following table describes the models of Table Store.

Model	Description
Wide Column model	The Wide Column model is applicable to various scenarios, such as metadata and big data. This model supports multiple functions, including data versions, time to live (TTL), auto-increment of primary key columns, conditional updates, local transactions, atomic counters, and filters.

Model	Description
Timeline model	The Timeline model is a data model that can meet special requirements of message data scenarios, such as message order preservation, storage of large numbers of messages, and real-time synchronization. This model also supports full-text queries and bool queries. The model is also suitable for use in scenarios such as instant messaging (IM) and feed streams.

Features

The following table describes the features of Table Store.

Feature	Description
#unique_10	If you set a primary key column as an auto-increment column , you do not need to enter values in this column when writing data in a row. Instead, Table Store automatically generates primary key values. The automatically generated key values are unique within the rows that share the same partition key. These values increase sequentially.
#unique_11	A conditional update is implemented only when specified conditions are met.
#unique_12	An atomic counter consists of columns. The atomic counter provides real-time statistics for some online applications, such as calculating the real-time page views (PVs) of a post.
#unique_13	Filters can be used to sort results on the server side. Only results that match the filtering conditions are returned. The feature effectively reduces the volume of transferred data and shortens the response time.
Search index	Based on inverted index and columnstore index, search-based index solves the complex query problem in big data scenarios.
Global secondary index	Global secondary index can be used to create indexes for attribute columns.
Tunnel Service	Tunnel Service provides tunnels that are used to export and consume data in the full, incremental, and differential modes . After creating tunnels, you can consume historical and incremental data exported from a specified table.
HBase support	Table Store HBase Client can be used to access Table Store through Java applications built on HBase APIs.

2 Limits

The following table describes the restrictions for Table Store. Some of the restrictions indicate the maximum values that can be used rather than the suggested values. In order to ensure better performance, please set the table structure and data size in a single row appropriately.

Item	Limit	Description
Number of instances created under an Alibaba Cloud user account	10	To increase the limit, open a ticket.
Number of tables in an instance	64	To increase the limit, open a ticket.
Instance name length	3-16 Bytes	Can contain uppercase and lowercase letters, digits, and hyphens. Must begin with a letter, and must not end with a hyphen. Must not contain the words, such as 'ali', 'ay', 'ots', 'taobao' and 'admin'.
Table name length	1-255 Bytes	Can contain uppercase and lowercase letters, digits, and underscores. Must begin with a letter or underscore.
Column name length	1-255 Bytes	Can contain uppercase and lowercase letters, digits, and underscores. Must begin with a letter or underscore.

Item	Limit	Description
Number of primary key columns	1-4 columns	Must be at least one column.
Size of string type primary key column values	1 KB	A single primary key column’ s string type column value is limited to 1 KB.
Size of string type attribute column values	2 MB	A single attribute column ’ s string type column value is limited to 2 MB.
Size of binary type primary key column values	1 KB	A single primary key column’ s binary type column value is limited to 1 KB.
Size of binary type attribute column values	2 MB	A single attribute column ’ s binary type column value is limited to 2 MB.
Number of attribute columns in a single row	Unlimited	A single row can contain an unlimited amount of attribute columns.
The number of attribute columns written by one request	1024 columns	The number of attribute columns written by one PutRow, UpdateRow, or BatchWriteRow request in a single row.
Data size of a single row	Unlimited	The total size of all column names, and column value data, for a single row is unlimited.
Reserved read/write throughput for a single table	0-5000	To increase the limit, open a ticket.
Number of columns in a read request’ s columns_to_get parameter	0-128	The maximum number of columns obtained in a row of data in the read request.
Table-level operation QPS	10	The QPS of a table-level operation on an instance must not exceed 10.

Item	Limit	Description
Number of UpdateTable operations for a single table	increase: Unlimited Lower : Unlimited	The reserved read/write throughput for each table can be increased or lowered unlimited times within a calendar day (from 00:00:00 to 00:00:00 of the next day in UTC time).
UpdateTable frequency for a single table	Maximum of one update every 2 minutes	The reserved read/write throughput for a single table cannot be adjusted beyond the frequency of once every 2 minutes.
The number of rows read by one BatchGetRow request	100	N/A
The number of rows written by one BatchWrite Row request	200	N/A
Data size of one BatchWrite Row request	4 MB	N/A
Data returned by one GetRange operation	5,000 rows or 4 MB	The data returned by a single operation cannot exceed 5000 rows or 4 MB . Otherwise, the excessive data will be read with a returned token.
The data size of an HTTP Request Body	5 MB	N/A

3 Terms

3.1 Instance

An instance is a logical entity in Table Store used to manage tables as a database in a relational database management system (RDBMS).

After activating Table Store, create an instance in the Table Store console and then create and manage tables within this instance. An instance is the basic unit in the resource management system of Table Store. Table Store implements access control and resource metering at the instance level.

You can create different instances for multiple businesses to manage their respective tables. You can also create multiple instances for one business based on different development, testing, and production purposes.

Table Store allows one Alibaba Cloud account to create up to 10 instances, and up to 64 tables can be created within each instance.

Naming rule

The name of each instance is unique within each region. You can create instances of the same names across different service regions. Naming rule for each instance must:

- Contain English letters, numbers, and hyphens(-)
- Start with English letters
- Not end with a hyphen(-)
- Be case-insensitive
- Be 3 Bytes to 16 Bytes in length
- Not contain the words, such as 'ali', 'ay', 'ots', 'taobao',and 'admin'

Instance type

Table Store supports two instance types: high-performance instance and capacity instance.

> **Notice:**
>
> An instance type cannot be modified once the instance is created.

The two instance types have the same functions and support petabyte-sized data volumes for a single table, however, they differ in costs and scenarios.

- High-performance instance

  High-performance instances support millions of read-write transactions per second (TPS) with 1 ms average latency of read and write operations per row. High-performance instances are suitable for scenarios requiring high read and write performance and concurrency, such as gaming, financial risk control, social networking applications, product recommendation systems, and public opinion sensing.
- Capacity instance

  Capacity instances provide write throughput and write performance comparable to that of the high-performance instances, but with lower costs. However, the capacity instances do not equal the read performance and concurrency of high-performance instances. The capacity instances are suitable for services with high write frequency but low read frequency, and services with high affordability and reduced performance requirements. This includes access to log monitoring data, Internet of Vehicles data, device data, time sequence data, and logistics data.

  Notice:

  Capacity instances do not support reserved read/write throughput. All reads and writes are billed based on the additional read/write throughput.

Instance type supported by region

Region Name	High-performance instance	Capacity instance
China East 1 (Hangzhou)	Supported	Supported
China East 2 (Shanghai)	Supported	Supported
China North 2 (Beijing)	Supported	Supported
China North 3 (Zhangjiakou)	In development	Supported
China North 5 (Huhehaote)	In development	Supported
China South 1 (Shenzhen)	Supported	Supported
China(Hong Kong)	In development	Supported

Region Name	High-performance instance	Capacity instance
Singapore	Supported	In development
US East 1 (Virginia)	Supported	In development
US West 1 (Silicon Valley)	Supported	In development
Asia Pacific NE 1 (Japan)	In development	Supported
Germany 1 (Frankfurt)	In development	Supported
Middle East 1 (Dubai)	In development	Supported
Asia Pacific SE 2 (Sydney)	In development	Supported
Asia Pacific SE 3 (Kuala Lumpur)	In development	Supported
Asia Pacific SE 5 (Jakarta)	In development	Supported
Asia Pacific SOU 1 (Mumbai)	In development	Supported

3.2 Endpoint

Each instance corresponds to an endpoint that is also known as the connection URL . The endpoint needs to be specified before any operations on the tables and data of Table Store.

- To access the data in Table Store from the Internet, the endpoint uses the following format:

```
https :// instanceNa  me . region . ots . aliyuncs . com
```

- To access the data in Table Store from an Alibaba Cloud ECS instance of the same region through the intranet, the endpoint uses the following format:

```
https :// instanceNa  me . region . ots - internal . aliyuncs . com
```

For example, to access the Table Store instance in China East 1 (Hangzhou) region, with the instance name of myInstance:

```
Endpoint   for   Internet   access :  https :// myInstance . cn -
hangzhou . ots . aliyuncs . com
```

```
Endpoint   for   intranet   access :  https :// myInstance . cn -
hangzhou . ots - internal . aliyuncs . com
```

Better performance, such as lower response latency and no unnecessary Internet traffic, can be expected through the intranet.

- If an application accesses Table Store from an ECS instance in VPC, the endpoint uses the following format:

```
https :// vpcName - instanceNa  me . region . vpc . ots . aliyuncs
. com
```

For example, the service address used by an application in China East 1 (Hangzhou) region to access the instance named myInstance from a network named testVPC:

```
Endpoint   of   VPC   access :  https :// testVPC - myInstance . cn
- hangzhou . vpc . ots . aliyuncs . com
```

This VPC access address is only used for access initiated by servers in the testVPC network.

3.3 Read/write throughput

The read/write throughput is measured by read/write capacity units (CUs), which is the smallest billing unit for the data read and write operations.

- One read CU indicates that 4 KB data is read from the table.
- One write CU indicates that 4 KB data is written into the table.
- Data smaller than 4 KB during the operation is rounded up to the nearest CU. For example, writing 7.6 KB data consumes two write CUs, and reading 0.1 KB data consumes one read CU.

When applications use an API to perform Table Store read/write operations, the corresponding amount of read/write CUs is consumed.

Reserved throughput

The reserved read/write throughput is an attribute of a table. When creating a table, the application specifies the read/write throughput reserved for the table. Configurin g the reserved read/write throughput does not affect the table’s access performance and service capability.

For reserved throughput billing, the reserved throughput value is always used to calculate the hourly fee even if an application consumes less than the specified amount of throughput.

For example, suppose that an application reads 3 KB of data per record and 80 records per second from a table. In this case, the application consumes 80 capacity units per second.

If you set the reserved read throughput to 80 capacity units per second, the hourly fee is calculated by using the following formula: Hourly Fee = 80 reserved read throughput capacity units x Hourly Price for Reserved Read Throughput. It is enough for 288000 (80 x 3600 seconds) reads per hour.

Note:

- Reserved read/write throughput can be set to zero.
- When the reserved read/write throughput is greater than zero, Table Store assigns and reserves enough resources for the table according to this configuration to guarantee low resource costs.
- For a non-zero reserved read/write throughput, your Table Store service is billed even if no read and write requests are made. To guarantee billing accuracy, Table Store limits the maximum reserved read/write throughput to 5000 CUs per table (neither read throughput nor write throughput can exceed 5000 CUs). If you require more than 5000 CUs of reserved read/write throughput for a single table, Open a ticket to increase the throughput.
- The reserved read/write throughput of a non-existent table is regarded as zero. To access a non-existent table, one additional read CU or one additional write CU is consumed depending on the actual operation.

Applications dynamically modify the reserved read/write throughput configuration of the table through the UpdateTable operation.

Additional throughput

The additional read/write throughput refers to the portion of the actual consumed read/write throughput that exceeds the reserved read/write throughput. Its refresh interval is one second.

In the following example, the reserved read throughput is set to 100 units. T0, T1, and T2 show the reserved read throughput and the additional read throughput that an application consumed in three consecutive seconds:

- T0: The actual read throughput consumption is 120 units. The consumption of the reserved read throughput and the consumption of the additional read throughput are 100 units and 20 units, respectively.
- T1: The actual read throughput consumption is 95 units. The consumption of the reserved read throughput and the consumption of the additional read throughput are 100 units and 0 units, respectively.
- T2: The actual read throughput consumption is 110 units. The consumption of the reserved read throughput and the consumption of the additional read throughput are 100 units and 10 units, respectively.

In the three consecutive seconds, the consumption of the reserved read throughput is 100 units, and the total consumption of the additional read throughput is 30 units.

Note:
Table Store uses the average value per hour to calculate the consumption of the reserved throughput and uses the total amount per hour to calculate the consumption of the additional throughput.

For the additional read/write throughput mode, it is difficult to estimate the amount of compute resources that need to be reserved for data tables. Table Store is required to provide sufficient service capability to effectively handle access traffic spikes. For this reason, the unit price of additional read/write throughput is higher than that of reserved read/write throughput. To make sure that low costs are maintained, we recommend that you set an appropriate value of the reserved read/write throughput.

Note:
Because it is difficult to accurately reserve resources based on the additional read/write throughput, in extreme situations, Table Store may return an error OTSCapacityUnitExhausted to an application when an access to a single partition key consumes 10,000 CUs per second. In this case, policies such as backoff retry are used to reduce the frequency of access to the table.

3.4 Region

Region refers to a service region of Alibaba Cloud.

Table Store is deployed across many service regions. You can select the most suitable region according to your requirements.

The following table lists the regions supported by Table Store.

Region Name	RegionID
China East 1 (Hangzhou)	cn-hangzhou
China East 2 (Shanghai)	cn-shanghai
China North 2 (Beijing)	cn-beijing
China North 3 (Zhangjiakou)	cn-zhangjiakou
China North 5 (Huhehaote)	cn-huhehaote
China South 1 (Shenzhen)	cn-shenzhen
China(Hong Kong)	cn-hongkong
Singapore	ap-southeast-1
US East 1 (Virginia)	us-east-1
US West 1 (Silicon Valley)	us-west-1
Asia Pacific NE 1 (Japan)	ap-northeast-1
Germany 1 (Frankfurt)	eu-central-1
Middle East 1 (Dubai)	me-east-1
Asia Pacific SE 2 (Sydney)	ap-southeast-2
Asia Pacific SE 3 (Kuala Lumpur)	ap-southeast-3
Asia Pacific SE 5 (Jakarta)	ap-southeast-5
Asia Pacific SOU 1 (Mumbai)	ap-south-1

4 Wide column model

4.1 Introduction

The Wide Column model differs from the relational model in the following aspects:

- The characteristics of Wide Column are: three-dimensional (rows, columns, and time), schema-free, wide columns, multi-version data, and TTL management.
- The characteristics of the relational model are : two-dimensional (rows and columns) and fixed schema.

The Wide Column model consists of the following parts:

- Primary key: Every row has a primary key with a multi-column structure (1-4 columns). The primary key is defined as a fixed schema, and is used primarily to uniquely distinguish a row of data.
- Partition key: The first column of the primary key is called a partition key. The partition key is used to partition the table by range. Every partition is distributively dispatched to services on different machines. Within the same partition key, we provide cross-row transactions. For more information, see Primary key and attribute.
- Attribute column: In one row, with the exception of the primary key, all other columns are attribute columns. Attribute columns correspond to many values. Different values correspond to different versions, and each row stores an unlimited number of attribute columns.
- Version: Each value corresponds to a different version that acts as a timestamp to define the time to live of that data.
- Data type: Table Store allows many different data types, including String, Binary, Double, Integer and Boolean.
- Time To Live (TTL): Each table defines the amount of time a data can be stored before being deleted. For example, if the TTL is defined as one month, the data written into the table more than a month ago will be cleared automatically. The write time of the data is determined by the version number. This write time is usually taken from the server time, but it can also be determined by the time

specified by the application. For more information, see Data versions and Time To Live.

- Max versions: Each table defines the maximum number of version data that can be stored in a column, which is used to control the number of versions in each column. If the number of versions in an attribute column exceeds the value in max versions, the earliest version is deleted.

4.2 Primary keys and attributes

In Table Store, tables, rows, primary keys, and attributes are the core components that you work with. A table is a collection of rows, and each row consists of a primary key and attributes. The first column of a primary key is called the partition key.

Primary keys

Primary keys are used to uniquely identify each row in a table. A primary key is a combination of one to four attributes. When creating a table, you must specify the composition of the primary key, including the name of each attribute, the data type of each attribute, and the sorted order of attributes. In Table Store, you can specify a data type, such as String, Binary, or Integer, for an attribute.

Table Store indexes data of a table based on the primary key of the table. All rows of the table are sorted in ascending or descending order based on the primary key.

Partition keys

The first column of a primary key is called the partition key. Table store assigns a row of data to the corresponding partitions determined by the range of each row's partition keys to achieve load balancing. Rows that have the same partition key value belong to the same partition. A partition may store rows with multiple partition key values. Table Store separates a partition or merges multiple partitions based on specific rules. This process is completed automatically.

Note:
The partition key is used as the minimum partition unit. Data under the same partition key value cannot split further. To prevent partitions from being too large to split, we recommend that the total size of all rows with the same partition key value is less than 10 GB.

Attributes

A row consists of multiple attributes. The number of attributes for each row is not restricted, which means that each row has a different number of attributes. The value of an attribute of a row can be null. The values of an attribute in multiple rows can be of different data types.

An attribute includes the version property. Multiple versions of attribute values can be retained as required for querying or other uses. Additionally, data in an attribute has its own TTL. For more information, see Data versions and life cycle.

4.3 Data versions and time to live

Version numbers

Each value of an attribute corresponds to a different version. The value of the version is the version number (timestamp). The version number is used to determine the Time to live (TTL).

When writing data, you are allowed to specify the version number of an attribute. If you do not specify a version number, the time from Jan 1, 1970, 00:00:00 UTC to the present time will be converted to milliseconds and used as the version number of the attribute. Version numbers are measured in milliseconds. When performing a comparison between TTL properties or Max Version Offset properties, you are required to divide version numbers by 1000 to convert the unit to seconds. The version number is used in the following scenarios:

- Time to live (TTL)

  The version number can be used to determine the lifecycle of a table. Assume that a version number of an attribute is 1468944000000, which is calculated based on the time of July 20, 2016, 00:00:00 UTC. When you set the TTL as 86400 (one day), the data of that version expires on July 21, 2016, 00:00:00 UTC. Then, the data is automatically deleted.

  When the version number of the data is determined by Table Store, the written data will be automatically cleansed after the specified TTL.

- Read the version number of each row's data

  When Table Store reads a row of data, you can specify the maximum number of versions or the range of version numbers of each attribute, which are allowed to be read.

Max Versions

When writing data, you can specify the version number of an attribute. The Max Versions property is used to determine how many versions of data of an attribute in a table can be retained. When the number of versions of an attribute exceeds the value of the Max Versions property, the data of the earliest version will be deleted asynchronously.

After creating a table, you are allowed to use the UpdateTable function to dynamically update the Max Versions property of the table.

> **Note:**
>
> - Data whose version exceeds the specified value of Max Versions is considered invalid. The data is neither visible to you nor being read, even if the data is not actually deleted.
> - Assume that you have decreased the value of Max Versions. When the number of versions exceeds the newly specified value of Max Versions, the earliest version will be deleted asynchronously.
> - Assume that you have increased the number of Max Versions. When the previous data whose version exceeds the previous value of Max Versions and has not been deleted, the data will be read.

Max Version Offset

The Max Version Offset property is used to determine the maximum allowed offset between the specified version number and the current system time. The property is measured in seconds. When the offset between the timestamp you have specified and the present time is greater than the specified TTL of a table, the written data expires immediately. You can set the Max Version Offset to prevent this situation.

To ensure that data is written successfully, Table Store will check the version number of an attribute when processing write requests. The range of valid version numbers of an attribute is: `[ The time when you write data - Max Version Offset , The time when you write data + Max Version`

`Offset`). The version number of an attribute is measured in milliseconds. After the version number is divided by 1000, the result that is measured in seconds must fall within this range. When a version number does not fall within the range, this write request fails.

Assume that the Max Version Offset property of a table is 86400 (one day). On July 21, 2016, 00:00:00 UTC, you are only allowed to write data whose version number is greater than 1468944000000, which is the result converted from July 20, 2016, 00:00:00 UTC, and less than 1469116800000, which is the result converted from July 22, 2016, 00:00:00 UTC. When the version number of an attribute in a row is 1468943999000, which is the result converted from July 19, 2016, 23:59:59 UTC, then the write request for the row fails.

Time to live

Time to live (TTL) is a property of a table. TTL is used to determine the lifecycle of the data. It is measured in seconds. To reduce storage costs, Table Store removes data that exceeds the specified TTL in the background to decrease your storage space.

Assume that the specified TTL of a table is 86400 (one day). On July 21, 2016, 00:00:00 UTC, attributes whose version numbers are less than 1468944000000 expire, which is the result converted from July 20, 2016, 00:00:00 UTC. Table Store will automatically remove the data of these attributes.

Note:

- Data that exceeds the specified TTL is invalid data. The data is neither visible to you nor being read, even if the data is not actually deleted.
- Assume that you decrease the TTL value. Some pieces of data will expire due to the decreased TTL value. The expired data is removed asynchronously.
- Assume that you increase the TTL value. If data that exceeds the previous TTL has not been removed, the data will be read again.

4.4 Naming conventions and data types

This topic describes the naming conventions and data types of Table Store.

Naming conventions

The following table describes naming conventions of tables and columns in Table Store.

Item	Description
Structure	A name can contain uppercase letters (A to Z), lowercase letters (a to z), digits (0 to 9), and underscores (_).
First character	A name must start with an uppercase letter (A to Z), a lowercase letter (a to z), or an underscore (_).
Case sensitivity	A name is case-sensitive.
Length	A name can be 1 to 255 characters in length.
Uniqueness	· A table name must be unique under the same instance. · Table names under different Table Store instances can be the same.

Data types of primary key columns

Data types of values in primary key columns include String, Integer, and Binary.

Data type	Description	Size limit
String	Data is in UTF-8. Empty strings are allowed.	Up to 1 KB
Integer	Data is 64-bit long.	Up to 8 Bytes
Binary	Data is binary. Empty values are allowed.	Up to 1 KB

Data types of attribute columns

The following table describes data types of values in attribute columns.

Data type	Description	Size limit
String	Data is in UTF-8. Empty strings are allowed.	For more information, see Limits.
Integer	Data is 64-bit long.	Up to 8 Bytes
Double	Data is 64-bit long.	Up to 8 Bytes
Boolean	The value can be True or False.	Up to 1 Byte
Binary	Data is binary. Empty values are allowed.	For more information, see Limits.

4.5 Auto-increment function of the primary key column

If you set a primary key column as an auto-increment column, you do not need to enter this column when writing data in a row. Instead, Table Store automatically generates the primary key value, which is unique in the partition key, and which increases progressively.

Features

Table Store, in conjunction with the auto-increment function of an primary key column, has the following features:

- The system architecture exclusive to Table Store and the implementation through an auto-increment primary key column make sure that the value generated for the auto-incrementing column is unique and strictly incrementing.
- The automatically generated auto-increment column value is a 64-bit signed long integer.
- The level of the partition key increases progressively.
- The auto-increment function is a table level. The tables with an auto-increment column and the tables without an auto-increment column can be created in the same instance.

If the auto-increment primary key column is set, the conditional update logic is not changed. See the following table for more information.

API	IGNORE	EXPECT_EXIST	EXPECT_NOT_EXIST
PutRow: The row exists.	Fail	Succeed	Fail
PutRow: The row does not exist.	Succeed	Fail	Fail
UpdateRow: The row exists.	Fail	Succeed	Fail
UpdateRow: The row does not exist.	Succeed	Fail	Fail
DeleteRow: The row exists.	Fail	Fail	Fail
DeleteRow: The row does not exist.	Succeed	Succeed	Fail

Limits

Table Store Auto-increment function of the primary key column mainly has the following restrictions:

- Table Store supports multiple primary keys. The first primary key is a partition key that cannot be set as an auto-increment column. However, one of other primary keys can be set as an auto-increment column.
- Only one primary key per table can be set as an auto-increment column.
- The attribute column cannot be set as an auto-increment column.
- The auto-increment column can only be set at the time the table is created. The existing table cannot set the auto-increment column.

Interface

- CreateTable
  - Set a column as an auto-incrementing column during table creation. For more information, see Primary key column auto-increment.
  - After table creation, you cannot configure the auto-incrementing feature of the table.
- UpdateTable

  You cannot change the auto-increment attribute of a table by using UpdateTable.
- PutRow/UpdateRow/BatchWriteRow
  - When writing the table, you do not need to set specific values for the column that you want to set as auto-incrementing. You only need to set a placeholder, for example, AUTO_INCREMENT. For more information, see Primary key column auto-increment.
  - You can set ReturnType in ReturnContent as RT_PK, that is, to return the complete primary key value, which can be used in the GetRow query.
- GetRow/BatchGetRow

  GetRow requires a complete primary key column, which can be obtained by setting ReturnType in PutRow, UpdateRow, or BatchWriteRow as RT_PK.
- Other interfaces

  Not changed

Usage

Java SDK: Auto-increment of the primary key column

Billing

The auto-increment function of primary key columns does not affect the existing billing logic. Returned data of the primary key column does not consume additional read CUs.

4.6 Conditional update

A conditional update is an update of table data that executes only when specified conditions are met. A conditional update can be based on a combination of up to 10 conditions. Supported conditions include arithmetic operations (=, ! =, >, >=, <, and <=) and logical operations (NOT, AND, and OR). The conditional update is applicable to PutRow, UpdateRow, DeleteRow, and BatchWriteRow.

The column-based judgment conditions include the row existence condition and column-based condition.

- The Row existence condition is classified into `IGNORE`, `EXPECT_EXIST`, and `EXPECT_NOT_EXIST`. When a table needs to be updated, the system first checks the row existence condition. If the row existence condition is not met, an error occurs during the update.
- The column-based condition supports `SingleColumnValueCondition` and `CompositeColumnValueCondition`, which are used to perform the condition-based judgment based on the values of a column or certain columns, similar to the conditions used by the Table Store filters.

Conditional update also supports optimistic locking strategy. That is, when a row needs to be updated, the system first obtains the value of a column. For example, the value of Column A is 1, and its condition is set as `Column A = 1`. Set `Column A = 2`, then update the row. If a failure occurs during the update, it means that the row has been successfully updated by another client.

Note:
In highly concurrent applications such as webpage view counting or gaming (where atomic counter updates are required), the probability of failed conditional updates is high. If this occurs, we recommend that you retry the update until successful.

Procedure

1. **Construct SingleColumnValueCondition.**

```
// set condition Col0 == 0 .
 SingleColu mnValueCon dition singleColu mnValueCon dition
= new SingleColu mnValueCon dition (" Col0 ",
        SingleColu mnValueCon dition . CompareOpe rator . EQUAL
, ColumnValu e . fromLong ( 0 ));
// If column Col0 does not exist , the condition
check fails .
 singleColu mnValueCon dition . setPassIfM issing ( false );
// Only check the latest version
 singleColu mnValueCon dition . setLatestV ersionsOnl y ( true
);
```

2. **Construct CompositeColumnValueCondition.**

```
// condition composite1 is ( Col0 == 0 ) AND ( Col1 >
100 )
 CompositeC olumnValue Condition composite1 = new
CompositeC olumnValue Condition ( CompositeC olumnValue
Condition . LogicOpera tor . AND );
 SingleColu mnValueCon dition single1 = new SingleColu
mnValueCon dition (" Col0 ",
        SingleColu mnValueCon dition . CompareOpe rator . EQUAL
, ColumnValu e . fromLong ( 0 ));
 SingleColu mnValueCon dition single2 = new SingleColu
mnValueCon dition (" Col1 ",
        SingleColu mnValueCon dition . CompareOpe rator .
GREATER_TH AN , ColumnValu e . fromLong ( 100 ));
 composite1 . addConditi on ( single1 );
 composite1 . addConditi on ( single2 );

// condition composite2 is ( ( Col0 == 0 ) AND ( Col1 >
100 ) ) OR ( Col2 <= 10 )
 CompositeC olumnValue Condition composite2 = new
CompositeC olumnValue Condition ( CompositeC olumnValue
Condition . LogicOpera tor . OR );
 SingleColu mnValueCon dition single3 = new SingleColu
mnValueCon dition (" Col2 ",
        SingleColu mnValueCon dition . CompareOpe rator .
LESS_EQUAL , ColumnValu e . fromLong ( 10 ));
 composite2 . addConditi on ( composite1 );
 composite2 . addConditi on ( single3 );
```

3. **Implement an increasing column by the optimistic locking strategy based on the conditional update.**

```
 private static void updateRowW ithConditi on ( SyncClient
  client , String pkValue ) {
    // construct the primary
     PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
     primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
     PrimaryKey primaryKey = primaryKey Builder . build ();

    // read a row
     SingleRowQ ueryCriter ia criteria = new SingleRowQ
ueryCriter ia ( TABLE_NAME , primaryKey );
```

```
        criteria . setMaxVers  ions ( 1 );
        GetRowResp  onse   getRowResp  onse  =  client . getRow ( new
     GetRowRequ  est ( criteria ));
        Row   row  =  getRowResp  onse . getRow ();
        long   col0Value  =  row . getLatestC  olumn (" Col0 ").
    getValue (). asLong ();

      //  Col0  =  Col0  +  1   by   conditiona  l   update
        RowUpdateC  hange   rowUpdateC  hange  =  new   RowUpdateC
    hange ( TABLE_NAME ,  primaryKey );
        Condition   condition  =  new   Condition ( RowExisten
    ceExpectat  ion . EXPECT_EXI  ST );
        ColumnCond  ition   columnCond  ition  =  new   SingleColu
    mnValueCon  dition (" Col0 ",  SingleColu  mnValueCon  dition .
    CompareOpe  rator . EQUAL ,  ColumnValu  e . fromLong ( col0Value
    ));
        condition . setColumnC  ondition ( columnCond  ition );
        rowUpdateC  hange . setConditi  on ( condition );
        rowUpdateC  hange . put ( new   Column (" Col0 ",  ColumnValu
    e . fromLong ( col0Value  +  1 )));

        try  {
            client . updateRow ( new   UpdateRowR  equest (
    rowUpdateC  hange ));
      }  catch  ( TableStore  Exception   ex ) {
            System . out . println ( ex . toString ());
      }
    }
```

Example

The following operations are examples of updates that are executed for highly concurrent applications:

```
//  Get   the   old   value
    old_value  =  Read ();
//  compute   such   as   increment   1
    new_value  =  func ( old_value );
//  Update   by   the   new   value
    Update ( new_value );
```

The conditional update makes sure `Update ( new_value )` if value equals to `old_value` in a highly concurrent environment where `old_value` may be updated by another client.

Billing

Writing or updating data successfully does not affect the capacity unit (CU) calculation rules of the interfaces. However, if the conditional update fails, one unit of write CU and one unit of read CU are consumed, which are billable.

4.7 Atomic counters

Atomic counter is a new feature of Table Store that allows you to implement an atomic counter on an attribute. This feature provides statistics data for online applications such as keeping track of the number of page views (PV) on various topics .

In traditional database systems (without atomic counters), you must perform read, modify, and write (RMW) operations to increment an attribute value by one or other number. You must read the previous attribute value from a database, and modify it on a client. Finally, you write the modified value to the database. The consistency issue occurs in a database while multiple clients modify data at the same time.

Currently, you can fix this issue by starting a transaction to lock a row. Then you can perform RMW operations in this transaction. You can use a transaction to ensure consistent data in a row when multiple clients modify a single row. However, this solution reduces write performance of atomic counters. RMW operations will increase network overhead.

To deal with increasing overhead, atomic counters are used in Table Store. A transaction within a sequence of RMW operations is sent to a database as a request . The database performs the operations on a row by locking the row. To ensure data consistency, you can update atomic counters on a database server to improve write performance.

Methods

The following methods are added in the RowUpdateChange class to operate an atomic counter:

- RowUpdateChange increment(Column column) is used to increment or decrement an attribute value by a number.
- void addReturnColumn(String columnName) is used to specify the name of an atomic counter that will be returned.
- void setReturnType(ReturnType.RT_AFTER_MODIFY) is used to specify a flag to indicate that the updated value of the atomic counter must be returned.

You can use RowUpdateChange to increment an atomic counter by a number as follows:

```
private   static   void   incrementB  yUpdateRow  Api ( SyncClient
client ) {
       //  You   can   specify   a   primary   key .
        PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
        primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString (" pk0 "));
        PrimaryKey   primaryKey  =  primaryKey  Builder . build ();

        RowUpdateC  hange   rowUpdateC  hange  =  new   RowUpdateC
hange ( TABLE_NAME ,  primaryKey );

       //  You   can   increment   the   price   value   by   10
without   specifying   a   timestamp .
        rowUpdateC  hange . increment ( new   Column (" price ",
ColumnValu  e . fromLong ( 10 )));

       //  You   can   specify   a   flag   to   indicate   that
the   updated   value   of   the   atomic   counter   must   be
returned .
        rowUpdateC  hange . addReturnC  olumn (" price ");
        rowUpdateC  hange . setReturnT  ype ( ReturnType .
RT_AFTER_M  ODIFY );

       //  You   can   update   the   price   attribute .
        UpdateRowR  esponse   response  =  client . updateRow ( new
UpdateRowR  equest ( rowUpdateC  hange ));

       //  You   can   display   the   updated   value .
        Row   row  =  result . getRow ();
        System . out . println ( row );
   }
```

Note:

- RowUpdateChange.addReturnColumn(an attribute name) is used to specify the name of an atomic counter that will be returned.
- RowUpdateChange.setReturnType(RT_AFTER_MODIFY is used to specify a flag to indicate that the updated value of the atomic counter must be returned.

Scenarios

You can use an atomic counter to keep track of a row in real time. Assume that you create a table to store pictures. Each row in the table has a user ID. An attribute of the row is used to store pictures. Another attribute of the row is used as an atomic counter to count the number of pictures.

- UpdateRow is used to add a picture to the table and increment the atomic counter by one.

- UpdateRow is used to remove a picture from the table and decrement the atomic counter by one.
- GetRow is used to read the value of the atomic counter to check the number of pictures.

This design ensures database consistency. When you add a picture to the table, the atomic counter is incremented by one instead of decremented by one.

Restrictions

Note the following restrictions when using atomic counters:

- Atomic counters only support the Integer type.
- The default value of an empty atomic counter is zero. When you implement an atomic counter on an existing attribute with a non-Integer type, an OTSParameterInvalid error occurs.
- You can update an atomic counter by using a positive or a negative number, but you must avoid an integer overflow. If an overflow issue appears, an OTSParameterInvalid error occurs.
- When you modify an atomic counter, the value will not be returned by default. You can use addReturnColumn() and setReturnType() to specify the name and updated value of an atomic counter that will be returned.
- You cannot update an attribute and an atomic counter simultaneously for a single request. If you have incremented or decremented the attribute A, then you cannot perform other operations, such as overwrite and delete operations on the attribute A.
- You can perform multiple update operations on the same row using a BatchWriteRow request. When you perform an atomic counter operation on a row, other operations in this BatchWriteRow request cannot be performed on this row.
- You can only implement an atomic counter on an attribute with the latest version. After you perform the update operation on the atomic counter, the atomic counter will be specified with a new version.
- An error may occur when an atomic counter encounters network timeouts or system failures. You can retry the operation. An atomic counter may be updated twice. This symptom leads to an overcounting or undercounting issue. In this case, we recommend that you can use conditional update to precisely update the attribute.

5 Timeline model

5.1 Introduction

Overview

The Timeline model is a data model designed for message data scenarios. The model supports some special requirements of message data scenarios, such as message order preservation, storage of large numbers of messages, and real-time synchronization. The model also supports the full-text search and bool query. The model is applicable to message scenarios such as instant messaging (IM) and Feed streams.

Architecture

The Timeline model provides clear core modules in a simple design. You can easily use this model, and set the model according to your business. The architecture of the model includes the following components:

- Store: a store of Timeline data. The store is similar to a table in a database.
- Identifier: an identifier used to identify Timeline data.
- Meta: the metadata used to describe Timeline data. The metadata is stored in a free-schema structure and can contain any column.
- Queue: stores all messages in a Timeline.
- SequenceId: the serial number of a message body in the Queue. The SequenceId values must be incremental and unique. The Timeline model generates SequenceId values by using an auto-increment column. You can also specify SequenceId values by manual.
- Message: the message body in the Timeline. The message is stored in a free-schema structure and can contain any column.
- Index: includes Meta Index and Message Index. You can customize indexes for any columns in Meta or Message to provide the bool query.

Features

The Timeline model supports the following features:

- Manages Meta data and messages, including basic data operations such as create, read, update, and delete.

- Supports the bool query and full-text search for Meta data and messages.
- Generates SequenceId values in two ways: auto-increment column and manual setting.
- Supports the Timeline Identifier that contains multiple columns.
- Compatible with the Timeline 1. X model. The TimelineMessageForV1 example of the Timeline model can directly read messages from and write messages to the V1 version.

Timeline

```
< dependency >
    < groupId > com . aliyun . openservic  es . tablestore </ groupId
 >
    < artifactId > Timeline </ artifactId >
    < version > 2 . 0 . 0 </ version >
</ dependency >
```

Table Store Java SDK (integrated with the Timeline model)

```
< dependency >
  < groupId > com . aliyun . openservic  es </ groupId >
  < artifactId > tablestore </ artifactId >
  < version > 4 . 12 . 1 </ version >
</ dependency >
```

5.2 Quick start

This topic describes how to get started with the Timeline model by using sample code.

Procedure

1. Log on to the Table Store console and create a Table Store instance. For more information, see #unique_46.
2. Download and install the Table Store Java SDK. For more information, see #unique_47.
3. Determine an endpoint and configure an AccessKey pair to initialize the instance. For more information, see #unique_48.
4. Download the sample code to get started with the Timeline model.

5.3 Basic operations

5.3.1 Initialization

Initialize the TimelineStore Factory

You can use SyncClient as a parameter to initialize the TimelineStore Factory and create a Store that manages Meta data and Timeline data. The retry operation after an error occurs depends on the retry policy of SyncClient. You can set SyncClient for the retry. If you have any special requirements, you can implement the RetryStrategy operation to customize the policy.

```
/**
 *  Set   the   retry   policy .
 *  Code :  configurat  ion . setRetrySt  rategy ( new   DefaultRet
 ryStrategy ());
 * */
 ClientConf  iguration   configurat  ion  =  new   ClientConf
 iguration ();

 SyncClient   client  =  new   SyncClient (
        " http :// instanceNa  me . cn - shanghai . ots . aliyuncs .
 com ",
        " accessKeyI  d ",
        " accessKeyS  ecret ",
        " instanceNa  me ",  configurat  ion );

 TimelineSt  oreFactory   factory  =  new   TimelineSt  oreFactory
 Impl ( client );
```

Initialize MetaStore

Create a schema for a Meta table. The schema includes parameters such as Identifier and MetaIndex. Create a Store that manages Meta data by using the TimelineStore Factory. You need to specify the following parameters: Meta table name, index, table name, primary key field, index name, and index type.

```
 TimelineId  entifierSc  hema   idSchema  =  new   TimelineId
 entifierSc  hema . Builder ()
        . addStringF  ield (" timeline_i  d "). build ();

 IndexSchem  a   metaIndex  =  new   IndexSchem  a ();
 metaIndex . addFieldSc  hema ( // Configure   the   index   field
 and   index   type .
         new   FieldSchem  a (" group_name ",  FieldType . TEXT ).
 setIndex ( true ). setAnalyze  r ( FieldSchem  a . Analyzer . MaxWord
 )
         new   FieldSchem  a (" create_tim  e ",  FieldType . Long ).
 setIndex ( true )
);

 TimelineMe  taSchema   metaSchema  =  new   TimelineMe  taSchema ("
 groupMeta ",  idSchema )
```

```
        . withIndex (" metaIndex ",  metaIndex ); // Set   the   index
 .

TimelineMe  taStore   timelineMe  taStore  =  serviceFac  tory .
createMeta  Store ( metaSchema );
```

Create a table

Create a table by using the parameters in metaSchema. Afterward, create and configure an index.

```
timelineMe  taStore . prepareTab  les ();
```

Delete a table

If a table contains an index, delete the index before deleting the table from the Store.

```
timelineMe  taStore . dropAllTab  les ();
```

Initialize TimelineStore

Create a schema for a Timeline table. The schema includes parameters such as Identifier and TimelineIndex. Create a Store that manages Timeline data by using the TimelineStore Factory. You need to specify the following parameters: Timeline table name, index, table name, primary key field, index name, and index type.

The BatchStore operation improves the concurrency performance on the basis of DefaultTableStoreWriter of Table Store. You can set the number of concurrent threads in the thread pool.

```
TimelineId  entifierSc  hema   idSchema  =  new   TimelineId
entifierSc  hema . Builder ()
       . addStringF  ield (" timeline_i  d "). build ();

IndexSchem  a   timelineIn  dex  =  new   IndexSchem  a ();
timelineIn  dex . setFieldSc  hemas ( Arrays . asList (// Configure
the   index   field   and   index   type .
        new   FieldSchem  a (" text ",  FieldType . TEXT ). setIndex
( true ). setAnalyze  r ( FieldSchem  a . Analyzer . MaxWord ),
        new   FieldSchem  a (" receivers ",  FieldType . KEYWORD ).
setIndex ( true ). setIsArray ( true )
));

TimelineSc  hema   timelineSc  hema  =  new   TimelineSc  hema ("
timeline ",  idSchema )
       . autoGenera  teSeqId () // Specify   the   auto - increment
column   as   the   method   to   generate   the   SequenceId   value
 .
       . setCallbac  kExecuteTh  reads ( 5 ) // Set   the   number
of   initial   threads   of   DefaultTab  leStoreWri  ter  to   5 .
       . withIndex (" metaIndex ",  timelineIn  dex ); // Set   the
index  .
```

```
TimelineSt  ore   timelineSt  ore  =  serviceFac  tory . createTime
lineStore ( timelineSc  hema );
```

Create a table

Create a table by using the parameters in TimelineSchema. Afterward, create and configure an index.

```
timelineSt  ore . prepareTab  les ();
```

Delete a table

If a table contains an index, delete the index before deleting the table from the Store.

```
timelineSt  ore . dropAllTab  les ();
```

5.3.2 Meta management

You can call some operations, such as Insert, Delete, Update, Read, and Search, to manage Meta data. The Search operation works on the basis of the Search Index feature. Only the MetaStore that has IndexSchema configured supports the Search operation. An index can be LONG, DOUBLE, BOOLEAN, KEYWORD, or GEO_POINT type. The index attributes include Index, Store, and Array, and have the same descriptions as those of the Search Index feature. For more information, see #unique_52.

Insert

The TimelineIdentifer value is used to identify Timeline data. Table Store overwrites repeated Identifier values.

```
TimelineId  entifier   identifier  =  new   TimelineId  entifier .
Builder ()
       . addField (" timeline_i  d ", " group ")
       . build ();
TimelineMe  ta   meta  =  new   TimelineMe  ta ( identifier )
       . setField (" filedName ", " fieldValue ");

timelineMe  taStore . insert ( meta );
```

Read

You can cal this operation to read TimelineMeta data in one row based on the Identifier value.

```
TimelineId  entifier   identifier  =  new   TimelineId  entifier .
Builder ()
       . addField (" timeline_i  d ", " group ")
       . build ();
```

```
timelineMe  taStore . read ( identifier );
```

Update

You can call this operation to update the Meta attribute that corresponds to the specified TimelineIdentifier value.

```
TimelineId  entifier   identifier  =  new   TimelineId  entifier .
Builder ()
       . addField (" timeline_i  d ", " group ")
       . build ();
TimelineMe  ta   meta  =  new   TimelineMe  ta ( identifier )
       . setField (" filedName ", " new   value ");

timelineMe  taStore . update ( meta );
```

Delete

You can call this operation to delete the TimelineMeta data in one row based on the Identifier value.

```
TimelineId  entifier   identifier  =  new   TimelineId  entifier .
Builder ()
       . addField (" timeline_i  d ", " group ")
       . build ();

timelineMe  taStore . delete ( identifier );
```

Search

You can call this operation to specify two search parameters: SearchParameter and the native SDK class SearchQuery. This operation returns Iterator<TimelineMeta>. You can iterate all result sets by using the iterator.

```
/**
 *  Search   meta   by   SearchPara  meter .
 * */
 SearchPara  meter   parameter  =  new   SearchPara  meter (
         field (" fieldName "). equals (" fieldValue ")
);
 timelineMe  taStore . search ( parameter );

/**
 *  Search   meta   by   SearchQuer  y .
 * */
 TermQuery   query  =  new   TermQuery ();
 query . setFieldNa  me (" fieldName ");
 query . setTerm ( ColumnValu  e . fromString (" fieldValue "));

 SearchQuer  y   searchQuer  y  =  new   SearchQuer  y (). setQuery (
 query );
```

```
timelineMe  taStore . search ( searchQuer  y );
```

5.3.3 Timeline management

You can call the operations for the fuzzy query and bool query to manage Timeline data. The query operations work on the basis of the Search Index feature. Only the TimelineStore that has IndexSchema configured supports the query operations. An index can be LONG, DOUBLE, BOOLEAN, KEYWORD, GEO_POINT, or TEXT type. The index attributes include Index, Store, Array, and Analyzer, and have the same descriptions as those of the Search Index feature. For more information, see #unique_52.

Search

You can call this operation to use the bool query. This query requires the field for a fuzzy query. You need to set the index type of the field to TEXT, and specify the tokenizer.

```
/**
 *  Search   timeline   by   SearchPara  meter .
 * */
 SearchPara  meter   searchPara  meter  =  new   SearchPara  meter (
         field (" text "). equals (" fieldValue ")
);
 timelineSt  ore . search ( searchPara  meter );

/**
 *  Search   timeline   by   SearchQuer  y .
 * */
 TermQuery   query  =  new   TermQuery ();
 query . setFieldNa  me (" text ");
 query . setTerm ( ColumnValu  e . fromString (" fieldValue "));
 SearchQuer  y   searchQuer  y  =  new   SearchQuer  y (). setQuery (
 query ). setLimit ( 10 );
 timelineSt  ore . search ( searchQuer  y );
```

Flush

The BatchStore operation works on the basis of the DefaultTableStoreWriter class in the SDK of Table Store. You can call the flush operation to trigger the process of sending the undelivered messages in the Buffer to Table Store and wait until Table Store stores all these messages.

```
/**
 *  Flush   messages   in   buffer ,  and   wait   until   all
 messages   are   stored .
 * */
```

```
timelineSt  ore . flush ();
```

5.3.4 Queue management

Obtain a Queue instance

A Queue is an abstract of a one message queue. The Queue corresponds to all messages of an identifier under a TimelineStore. You can call the required operation of TimelineStore to create a Queue instance.

```
TimelineId  entifier   identifier  =  new   TimelineId  entifier .
Builder ()
       . addField (" timeline_i  d ", " group_1 ")
       . build ();

//  The   Queue   correspond  s   to   an   identifier   of   a
TimelineSt  ore .
TimelineQu  eue   timelineQu  eue  =  timelineSt  ore . createTime
lineQueue ( identifier );
```

The Queue instance manages a message queue that corresponds to an identifier of a TimelineStore. This instance provides some operations, such as Store, StoreAsync, BatchStore, Delete, Update, UpdateAsync, Get, and Scan.

Store

You can call this operation to synchronously store messages. To use this operation, you can set SequenceId in two ways: auto-increment column and manual setting.

```
timelineQu  eue . store ( message );// Auto - increment   column
timelineQu  eue . store ( sequenceId ,  message );// Manual   setting
```

StoreAsync

You can call this operation to asynchronously store messages. You can customize callbacks to process successful or failed storage. This operation returns Future<TimelineEntry>.

```
TimelineCa  llback   callback  =  new   TimelineCa  llback () {
   @ Override
    public   void   onComplete  d ( TimelineId  entifier   i ,
TimelineMe  ssage   m ,  TimelineEn  try   t ) {
       //  do   something   when   succeed .
   }

   @ Override
    public   void   onFailed ( TimelineId  entifier   i ,
TimelineMe  ssage   m ,  Exception   e ) {
       //  do   something   when   failed .
   }
};
```

```
timelineQu  eue . storeAsync ( message ,  callback );// Generate
  the   SequenceId   value   by   using   an   auto - increment
column .
timelineQu  eue . storeAsync ( sequenceId ,  message ,  callback );//
Specify   the   SequenceId   value   by   manual .
```

BatchStore

You can call this operation to store multiple messages in the callback and non-callback ways. You can customize callbacks to process successful or failed storage.

```
timelineQu  eue . batchStore ( message );// Auto - increment
column
timelineQu  eue . batchStore ( sequenceId ,  message );// Manual
setting

timelineQu  eue . batchStore ( message ,  callback );// Auto -
increment   column
timelineQu  eue . batchStore ( sequenceId ,  message ,  callback );//
Manual   setting
```

Get

You can call this operation to read one row based on the SequenceId value. If the message does not exist, no error occurs and the system returns null.

```
timelineQu  eue . get ( sequenceId );
```

GetLatestTimelineEntry

You can call this operation to read the latest message. If the message does not exist, no error occurs and the system returns null.

```
timelineQu  eue . getLatestT  imelineEnt  ry ();
```

GetLatestSequenceId

You can call this operation to obtain the SequenceId value of the latest message. If the message does not exist, no error occurs and the system returns 0.

```
timelineQu  eue . getLatestS  equenceId ();
```

Update

You can call this operation to synchronously update a message based on the SequenceId value.

```
TimelineMe  ssage   message  =  new   TimelineMe  ssage (). setField
(" text ", " Timeline   is   fine .");

// update   message   with   new   field
message . setField (" text ", " new   value ");
```

```
timelineQu  eue . update ( sequenceId ,  message );
```

UpdateAsync

You can call this operation to asynchronously update a message based on the SequenceId value. You can customize callbacks to process a successful or failed update. This operation returns Future<TimelineEntry>.

```
TimelineMe  ssage   oldMessage  =  new   TimelineMe  ssage ().
setField (" text ", " Timeline   is   fine .") ;
TimelineCa  llback   callback  =  new   TimelineCa  llback () {
   @ Override
    public   void   onComplete  d ( TimelineId  entifier   i ,
TimelineMe  ssage   m ,  TimelineEn  try   t ) {
       //  do   something   when   succeed .
   }

   @ Override
    public   void   onFailed ( TimelineId  entifier   i ,
TimelineMe  ssage   m ,  Exception   e ) {
       //  do   something   when   failed .
   }
};

TimelineMe  ssage   newMessage  =  oldMessage ;
newMessage . setField (" text ", " new   value ");
timelineQu  eue . updateAsyn  c ( sequenceId ,  newMessage ,
callback );
```

Delete

You can call this operation to delete one row based on the SequenceId value.

```
timelineQu  eue . delete ( sequenceId );
```

Scan

You can call this operation to read messages in one queue in forward or backward order based on the Scan parameter. This operation returns Iterator<TimelineEntry>. You can iterate all result sets by using the iterator.

```
ScanParame  ter   scanParame  ter  =  new   ScanParame  ter ().
scanBackwa  rd ( Long . MAX_VALUE ,  0 );

timelineQu  eue . scan ( scanParame  ter );
```

6 Search Index

6.1 Overview

You can use the multiple efficient index schemas of search index to solve complex query problems in big data scenarios.

A table in Table Store is a distributed NoSQL data schema. Such tables can support storage and read/write of large-scale data, such as monitoring data and log data. Originally, Table Store only supports queries based on primary key columns, such as reading data in a single row and within a specified range. Other types of queries were not available, such as queries based on non-primary key columns and the bool query.

To resolve this issue, Table Store has provided the search index feature. Based on inverted indexes and column-oriented storage, search index supports multiple queries, including but not limited to:

- Query based on non-primary key columns
- Bool query
- Full-text search
- Query by geographical location
- Prefix query
- Fuzzy query
- Nested query

Index differences

Aside from queries based on primary key columns in the primary table, Table Store provides two index schemas for accelerated queries: global secondary index and search index. The following table describes the differences among the three indexes.

Index type	Description	Scenario
Table	A table is similar to a big map . Tables only support queries based on primary key columns.	· You can specify the complete primary key columns. · You can specify the prefixes of primary key columns.

Index type	Description	Scenario
Global secondary index	You can create one or more global secondary indexes and issue query requests against these indexes. This way, you can perform queries based on the primary key columns of these indexes.	· You can determine the required columns in advance, and only a few columns are required. · You can specify the complete primary key columns or the prefixes of primary key columns.
Search index	Search index uses inverted indexes, Bkd-trees, and column-oriented storage to meet various query scenarios.	All query and analysis scenarios that the table and the global secondary index do not support.

Precautions

Index synchronization

If you have created a search index for a table, data is written to the table first. When the write is successful, success message is immediately returned to the user. At the same time, another asynchronous thread reads the newly written data from the table and writes the data to the search index. This is an asynchronous process.

The asynchronous data synchronization between a table and search index does not affect the write performance of Table Store. The indexing latency is within seconds, most of which are within 10 seconds. You can view the indexing latency in the Table Store console in real time.

TTL

You cannot create a search index in a table where you have specified the time to live (TTL) parameter.

max versions

You cannot create a search index in a table where you have specified the max versions parameter.

You can customize the timestamp whenever you write data to an attribute column that allows only one version. If you first write a major version number and then a minor version number, the index of the major version number may be overwritten by the index of the minor version number.

Features

Search index can solve complex query problems in big data scenarios. Other systems such as databases and search engines can also solve data query problems. The differences between Table Store and databases and search engines are illustrated as follows:

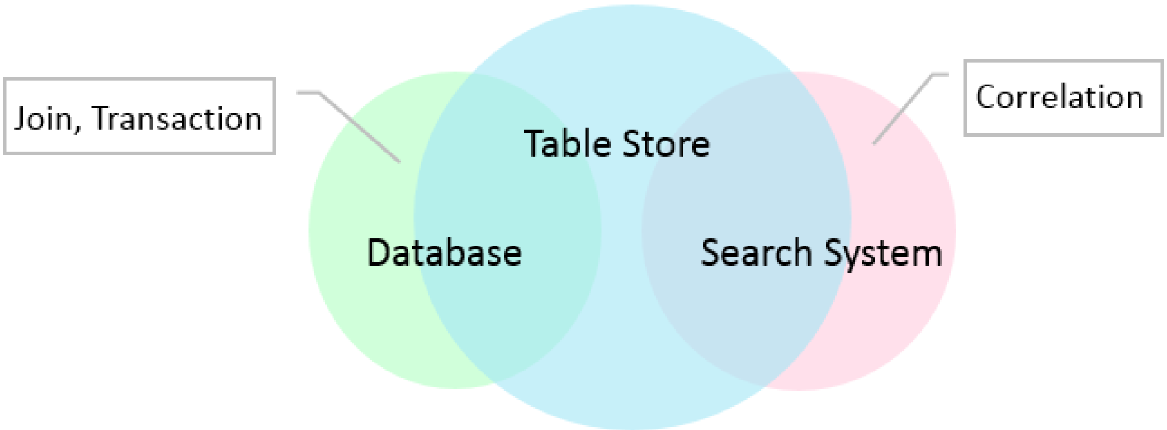


Table Store can provide all features of databases and search engines, except for join operations, transactions, and relevance of search results. Table Store also has high data reliability of databases and supports advanced queries of search engines. Therefore, Table Store can replace the common `database plus search engine architectu re`. If you do not need join operations, transactions, and relevance of search results, we recommend that you use search index of Table Store.

6.2 Features

This topic describes the core features of search index.

Core features

Query based on non-primary key columns

Originally, Table Store only supports queries based on complete primary key columns or their prefixes. Queries based on non-primary key columns were not available in some scenarios. Search index enables Table Store to support queries based on non-primary key columns. You only need to create a search index for the column to be queried.

Bool query

Bool query is applicable to order scenarios. In order scenarios, a table may contain dozens of fields. You cannot determine how to combine fields required for queries when you create a table. Even if the combination of required fields is specified, hundreds of combinations may be available. If you use a relational database, you need to create hundreds of indexes. In addition, if a certain combination is not created in advance, you cannot query the corresponding data.

However, you can use Table Store to create a search index that includes the required field names, which can be combined in a query as needed. Search index also supports multiple logical operators, such as AND, OR, and NOT.

Query by geographical location

With the popularization of mobile devices, geographical location data is becoming increasingly important. The data is used in most apps, such as WeChat Moments, Sina Weibo, food delivery apps, sports apps, and Internet of Vehicles (IoV) apps. These apps provide geographical location data. Therefore, they must support query features .

Search index supports queries based on the following geographical location data:

- Near: queries points within a specified radius based on a central point.
- Within: queries points within a specified rectangular or polygonal area.

Based on these query features, you can use Table Store to easily query geographical location data without resorting to other databases or search engines.

Full-text search

Search index can tokenize data to perform full-text search. However, unlike search engines, Table Store cannot return relevant results in response to a query. Therefore, if you need relevant results, we recommend that you use search engines.

Five tokenization types are available, including single-word tokenization, delimiter tokenization, minimum semantic unit-based tokenization, maximum semantic unit-based tokenization, and fuzzy tokenization. For more information, see #unique_58.

Fuzzy query

Search index supports queries based on wildcards. This feature is similar to the LIKE operator in relational databases. You can specify characters and wildcards such as question marks (`?`) or asterisks (`*`) to query data in the way similar to the LIKE operator.

Prefix query

Search index supports the prefix query feature. This feature is applicable to any natural language. For example, in the query based on the prefix "apple", the system may return words such as "apple6s" and "applexr".

Nested query

In addition to a flat structure, online data such as labeled pictures have some complex multilayered structures. For example, a database stores a large number of pictures , and each picture has multiple elements, such as houses, cars, and people. Each element in a picture has a unique score. The score is evaluated based on the size and position of the element in a picture. Therefore, each picture has multiple labels. Each label has a name and a weighted score. You can use nested queries based on the conditions or field names of the labels.

The following example shows the JSON data format in a query:

```
{
 " tags ": [
   {
      " name ": " car ",
      " score ":  0 . 78
   },
   {
      " name ": " tree ",
      " score ":  0 . 24
   }
 ]
}
```

You can use the nested query effectively to store and query data of multilayered logical relationships. This query facilitates the modeling of complex data.

Deduplication

Search index supports deduplication for query results. Deduplication allows you to specify the highest frequency of occurrence of an attribute value to achieve high cardinality. For example, when you search for a laptop on an e-commerce platform , the first page may display products of a certain brand. This is not a user-friendly result. However, the deduplication feature of Table Store can resolve this issue.

Sorting

A table sorts data based on the alphabetical order of primary key columns. To sort data by other fields, you need to use the sorting feature of search index. Table Store supports multiple types of sorting, such as ascending sorting, descending sorting, single-field sorting, and multi-field sorting. By default, Table Store returns results based on the order of primary key columns. You can use this method to sort global data.

Total number of rows

You can specify the number of rows that the system returns for the current request when you use search index for a query. If you do not specify any query condition for search index, the system returns the total number of rows where you have created indexes. When you stop writing new data to a table and create indexes on all attributes, the system returns the total number of rows in the table. This feature applies to data verification and data management.

SQL

Table Store does not support SQL statements and operators. However, most of these SQL features can match similar features of search index, as shown in the following table.

SQL	Search index	Supported
SHOW	API operation: DescribeSearchIndex	Yes
SELECT	Parameter: ColumnsToGet	Yes
FROM	Parameter: index name	Supported for single indexes and not supported for multiple indexes
WHERE	Query: a variety of queries such as TermQuery	Yes
ORDER BY	Parameter: sort	Yes
LIMIT	Parameter: limit	Yes
DELETE	API operation: query followed by DeleteRow	Yes
LIKE	Query: wildcard query	Yes
AND	Parameter: operator = and	Yes
OR	Parameter: operator = or	Yes
NOT	Query: bool query	Yes
BETWEEN	Query: range query	Yes
NULL	ExistQuery	Yes

6.3 API operations

6.3.1 Overview

This topic describes the operations, fields, queries, and billing methods of search index.

SDKs

You can use the following SDKs to implement search index.

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

API operations

Action	Operation	Description
Create	CreateSearchIndex	Creates a search index.
Describe	DescribeSearchIndex	Queries detailed information of a search index.
List	ListSearchIndex	Queries the list of search indexes.
Delete	DeleteSearchIndex	Deletes a specified search index.
Search	Search	Searches for required data.

Fields

The value of a search index field in Table Store is the value of the field of the same name in the corresponding table. The types of these fields must match each other, as described in the following table.

Field type in the search index	Field type in the table	Description
Long	Integer	64-bit long integers.

Field type in the search index	Field type in the table	Description
Double	Double	64-bit long floating-point numbers.
Boolean	Boolean	Boolean values.
Keyword	String	Character strings that cannot be tokenized.
Text	String	Character strings or text that can be tokenized. For more information, see #unique_58.
Geopoint	String	Geographical coordinates in the `latitude ,` `longitude` format. Example: 35.8,-45.91.
Nested	String	Nested type fields, such as "{["a": 1], ["a": 3]}."

Notice:

The types in this table must correspond to each other. Otherwise, Table Store discards the data as dirty data. Make sure fields of the Geopoint and Nested types must comply with the formats described in the preceding table. If the formats do not match, Table Store discards the data as dirty data. As a result, the data may be available in the table, but be unavailable in the search index.

Aside from the type attribute, search index fields also have additional attributes.

Attribute	Type	Option	Description
Index	Boolean	Specifies whether to create an index for a column.	· True indicates that Table Store creates an inverted index or spatial index for the column. · False indicates that Table Store does not create an index for the column. · If no indexes exist, you cannot query by the column.
EnableSortAndAgg	Boolean	Specifies whether to enable sorting and aggregation.	· True indicates that data can be sorted by using the column. · False indicates that data cannot be sorted by using the column.
Store	Boolean	Specifies whether to store original values in the index.	True indicates that Table Store stores the original values in the column to the index. Therefore, Table Store reads values of the column directly from the index, rather than from the primary table. This optimizes query performance.

Attribute	Type	Option	Description
IsArray	Boolean	Specifies whether the column is an array.	· True indicates that the column is an array. Data written to the column must be a JSON array, such as ["a","b","c"]. · You do not need to explicitly specify this parameter for Nested columns because they are arrays. · Array type data can be used in all queries because arrays do not affect queries.

For more information about the attributes that each field type supports, see the following table.

Type	Index	EnableSortAndAgg	Store	Array
Long	Supported	Supported	Supported	Supported
Double	Supported	Supported	Supported	Supported
Boolean	Supported	Supported	Supported	Supported
Keyword	Supported	Supported	Supported	Supported
Text	Supported	Not supported	Supported	Supported
Geopoint	Supported	Supported	Supported	Supported
Nested	Required for child fields.	Required for child fields.	Required for child fields.	Nested fields are arrays.

Query parameters and types

You must specify SearchRequest in a query. The following table describes parameters that are included in SearchRequest.

Parameter	Data type	Description
offset	Integer	Specifies the position from which the current query starts.
limit	Integer	Specifies the maximum number of items that the current query returns.
getTotalCount	Boolean	Specifies whether to return the total number of matched rows. This parameter is set to false by default. A value of true may affect the query performance.
Sort	Sort	Specifies the field and method for sorting.
collapse	Collapse	Specifies the name of the field that you want to collapse in the query result.
query	Query	Specifies the type of the current query. The following table lists the query types.

Name	Query	Description
Query by matching all rows	MatchAllQuery	You can use MatchAllQuery to check the total number of rows.
Query by tokenized data	MatchQuery	You can use MatchQuery to tokenize the query data, and query the tokenized data. Logical operator OR applies to tokens.
Query by matched phrases	MatchPhraseQuery	This query is similar to MatchQuery. The matched tokens must be adjacent to each other in the query data.

Name	Query	Description
Query by exact match	TermQuery	You can use TermQuery to match exact strings. Table Store uses exact matches to query data in a table, and does not tokenize the query data.
Query by multiple terms	TermsQuery	This query is similar to TermQuery. You can use TermsQuery to match multiple terms, which is similar to the SQL `IN` operator.
Query by prefix	PrefixQuery	You can use PrefixQuery to query data in a table by matching a specified prefix .
Query by range	RangeQuery	You can use RangeQuery to query data within a specified range in a table.
Query by wildcards	WildcardQuery	You can use WildcardQu ery to query data based on strings that contain one or more wildcards. This query is similar to the SQL LIKE operator.
Query by a combination of filtering conditions	BoolQuery	You can use BoolQuery to combine multiple filtering conditions by using Logical operators, such as AND, OR, and NOT.
Query by matching data within a rectangular geographical area	GeoBoundingBoxQuery	You can use GeoBoundin gBoxQuery to specify a rectangular geographic al area as a filtering condition in a query. Table Store returns the rows where the value of a field falls within the rectangula r geographical area.

Name	Query	Description
Query by matching data within a circular geographical area	GeoDistanceQuery	You can use GeoDistanc eQuery to specify a circular geographical area as a filtering condition in a query, which consists of a central point and radius . Table Store returns the rows where the value of a field falls within the circular geographical area.
Query by matching data within a polygonal geographical area	GeoPolygonQuery	You can use GeoPolygon Query to specify a polygonal geographic al area as a filtering condition in a query. Table Store returns the rows where the value of a field falls within the polygonal geographical area.

Pricing

For more information, see #unique_84.

6.3.2 CreateSearchIndex

You can call this operation to create a search index. To use the search index feature for a table, you must create a search index in the table. One table can contain multiple search indexes.

You can call the Search operation to query fields (including primary key columns and attribute columns) included in the search index.

Description

Parameters:

- TableName: specifies the name of the table for which you want to create a search index.
- IndexName: specifies the name of the search index.

- IndexSchema: defines the schema of the search index.
  - IndexSetting
    - RoutingFields: specifies the routing fields. You can specify some primary key columns as routing fields. Table Store distributes data that is written to a search index to different partitions based on the specified routing fields. The data with the same routing field values is distributed to the same data partition.
  - FieldSchemas
    - FieldName: required. This parameter specifies the name of the field that is a column name in the table. The name is of the string type.
    - FieldType: required. This parameter specifies the type of the field. For more information, see the "Fields" section in Overview.
    - Index: optional. This parameter specifies whether to create an index for the field. The index is of the Boolean type. Default value: true.
    - IndexOptions: optional. This parameter specifies whether to store terms such as position and offset in an inverted list. Use the default value in general conditions.
    - EnableSortAndAgg: optional. This parameter specifies whether to enable sorting and aggregation. This parameter is of the Boolean type. Default value : true.
    - Store: optional. This parameter specifies whether to store original values in the index to accelerate queries. This parameter is of the Boolean type. Default value: true.

FAQ

How many indexes can be created in a table?

Assume that you have a table with five fields: ID, name, age, city, and sex, and you need to query by name, age, or city. There are two methods to create search indexes:

- Method 1: Create a search index for an index field

  In this case, you need to create three search indexes: name_index, age_index, and city_index. You can use city_index to query data by city, and age_index to query data by age.

  However, you cannot use this method to query students who are younger than 12 years old and live in Chengdu.

  The implementation of this method is similar to that of secondary indexes. In this case, one index field for one search index brings no benefits to search indexing but increases costs. Therefore, we recommend that you do not use this method to create a search index.
- Method 2: Create a search index for multiple index fields

  In this case, you only need to create a search index named student_index. The fields include name, age, and city. You can use the city index field in the student_index to query data by city. You can use the age index field in the student_index to query data by age.

  You can use the age and city index fields in the student_index to query students who are younger than 12 years old and live in Chengdu.

  This method provides more functions at low cost. We recommend that you use this method.

Limits

1. Timeliness of index creation

It takes a few minutes to create a search index. During the creation process, you can write data into the table.

2. Quantity

For more information, see #unique_86.

Examples

```
/**
 * Create  a  search  index  that  contains  the  Col_Keywor
 d  and  Col_Long  columns . Set  the  type  of  data  in
 Col_Keywor d  to  KEYWORD . Set  the  type  of  data  in
 Col_Long  to  LONG .
 */
 private  static  void  createSear chIndex ( SyncClient  client )
 {
```

```
    CreateSear  chIndexReq  uest   request  =  new   CreateSear
chIndexReq  uest ();
    request . setTableNa  me ( TABLE_NAME ); //  Set   the   table
name .
    request . setIndexNa  me ( INDEX_NAME ); //  Set   the   index
name .
    IndexSchem  a   indexSchem  a  =  new   IndexSchem  a ();
    indexSchem  a . setFieldSc  hemas ( Arrays . asList (
            new   FieldSchem  a (" Col_Keywor  d ",  FieldType .
KEYWORD ) //  Set   the   field   name   and   field   type .
                   . setIndex ( true ) //  Set   the   parameter   to
  true   to   enable   indexing .
                   . setEnableS  ortAndAgg ( true ), //  Set   the
parameter   to   true   to   enable   sorting   and   aggregatio  n .
            new   FieldSchem  a (" Col_Long ",  FieldType . LONG )
                   . setIndex ( true )
                   . setEnableS  ortAndAgg ( true )));
    request . setIndexSc  hema ( indexSchem  a );
    client . createSear  chIndex ( request ); //  Use   the   client
  to   create   a   search   index .
}
```

6.3.3 DescribeSearchIndex

You can call this operation to query the details of a Search Index structure. To use the Search Index feature for a table, you must create a Search Index structure in the table. One table can contain multiple Search Index structures.

Description

Name: DescribeSearchIndex

Parameters:

- TableName: the name of the target table where you request the details of the Search Index structure.
- IndexName: the name of the target index.

Example

```
private   static   DescribeSe  archIndexR  esponse   describeSe
archIndex ( SyncClient   client ) {
    DescribeSe  archIndexR  equest   request  =  new   DescribeSe
archIndexR  equest ();
    request . setTableNa  me ( TABLE_NAME ); //  Set   the   name
of   the   table .
    request . setIndexNa  me ( INDEX_NAME ); //  Set   the   name
of   the   index .
    DescribeSe  archIndexR  esponse   response  =  client .
describeSe  archIndex ( request );
    System . out . println ( response . jsonize ()); //  Display
the   details   of   the   response .
    return   response ;
```

```
}
```

6.3.4 ListSearchIndex

You can call this operation to retrieve the list of all Search Index structures associated with an instance or a table.

Description

Name: ListSearchIndex

Parameter:

- TableName: the name of the target table. If you do not specify this optional parameter, Table Store returns the list of all indexes on the instance. If you specify a table, Table Store returns the list of all Search Index structures associated with the table.

Example

```
private   static   List < SearchInde  xInfo >  listSearch  Index (
SyncClient   client ) {
    ListSearch  IndexReque  st   request  =  new   ListSearch
IndexReque  st ();
    request . setTableNa  me ( TABLE_NAME ); //  Set   the   name
of   the   table .
    return   client . listSearch  Index ( request ). getIndexIn
fos (); //  Return   all   Search   Index   structures   of   the
specified   table .
}
```

6.3.5 DeleteSearchIndex

You can call this operation to delete a Search Index structure.

Description

Name: DeleteSearchIndex

Parameters:

- TableName: the name of the target table where you delete the Search Index structure.
- IndexName: the name of the target index that you want to delete.

Example

```
private   static   void   deleteSear  chIndex ( SyncClient   client )
{
    DeleteSear  chIndexReq  uest   request  =  new   DeleteSear
chIndexReq  uest ();
```

```
    request . setTableNa  me ( TABLE_NAME ); //  Set   the   name
 of   the   table .
    request . setIndexNa  me ( INDEX_NAME ); //  Set   the   name
 of   the   index .
    client . deleteSear  chIndex ( request ); //  Use   client   to
 delete   the   target   Search   Index   structure .
}
```

6.3.6 Array and Nested field types

Aside from basic field types, such as Long, Double, Boolean, Keyword, Text, and Geopoint, search index also provides two special field types.

One is the Array type. The Array type can be attached to the basic field types. For example, a field of Long type plus an Array type forms an integer array. This field can contain multiple long integers. If any data of a row is matched in the query, the row is returned.

The other is the Nested type, which provides more features than the Array type.

Array type

Basic Array types, such as:

- Long Array: an array of long integers. Format: "[1000, 4, 5555]."
- Boolean Array: an array of Boolean values. Format: "[true, false]."
- Double Array: an array of floating-point numbers. Format: "[3.1415926, 0.99]."
- Keyword Array: an array of strings.
- Text Array: an array of text. This type is not common.
- GeoPoint Array: an array of geographical locations. Format: "[34.2, 43.0], [21.4, 45.2]."

The Array type is only supported in search index. Therefore, when the type of an index field involves Array, the field in the table must be of the String type. The basic data type in the search index remains, such as Long or Double. For example, when a price field is of the Double Array type, the field must be of the String type in the table, and of the Double type in the search index, with isArray set to true.

Nested type

A Nested column contains nested documents. One document or one row can contain multiple child documents, and these child documents are saved to the same Nested column. You need to specify the schema of child documents in the Nested column.

The structure includes the fields of the child documents and the property of each field . The following example defines the format of a Nested column in Java:

```
// Specify the FieldSchem a class for the child
 documents .
 List < FieldSchem a > subFieldSc hemas = new ArrayList <
 FieldSchem a >();
 subFieldSc hemas . add ( new FieldSchem a (" tagName ",
 FieldType . KEYWORD )
    . setIndex ( true ). setEnableS ortAndAgg ( true ));
 subFieldSc hemas . add ( new FieldSchem a (" score ", FieldType
 . DOUBLE )
    . setIndex ( true ). setEnableS ortAndAgg ( true ));

// Set FieldSchem a of the child documents as
 subfieldSc hemas of the Nested column .
 FieldSchem a nestedFiel dSchema = new FieldSchem a (" tags
 ", FieldType . NESTED )
    . setSubFiel dSchemas ( subFieldSc hemas );
```

This example defines the format of a Nested column named tags. The child documents include two fields: one is a KEYWORD field named tagName and the other is a DOUBLE field named score.

Table Store writes Nested columns as strings in JSON arrays to the table. The following example shows the data format of a Nested column:

```
[{" tagName ":" tag1 ", " score ": 0 . 8 }, {" tagName ":" tag2 ", "
 score ": 0 . 2 }]
```

This column contains two child documents. Even if a column contains only one child document, you must provide the strings in JSON arrays.

The Nested type has the following limits:

1. Nested indexes do not support the IndexSort feature. However, IndexSort can improve query performance in many scenarios.
2. The nested query provides lower performance than other types of queries.

Apart from the preceding limits, the Nested type supports all queries and sorting, and will support statistical aggregation in the future.

6.3.7 Sort

You can use Sort to specify the method of sorting the result when you call the Search operation to search indexes.

The Search Index feature supports multiple sorting methods.

If you have not specified the sorting method for the search, the system applies the IndexSort parameter for the required indexes. By default, Table Store returns the query result in the order of primary key columns.

Table Store supports the following sorting methods:

- ScoreSort

  Sort the result by relevance score. ScoreSort is applicable to relevance scenarios such as full-text indexing.
- PrimaryKeySort

  Sort the result by the value of a primary key.
- FieldSort

  Sort the result by the value of a specified field.
- GeoDistanceSort

  Sort the result by the distance, radius, from a central point.

6.3.8 Tokenization

Search index can tokenize words for queries. If the field type is set to text, you can set an additional tokenization parameter for this field to specify the method in which the text is tokenized. Tokenization cannot be set for fields of non-text types.

You can use MatchQuery and MatchPhraseQuery to query text data. TermQuery, TermsQuery, PrefixQuery, and WildcardQuery are also used in a few scenarios.

The following tokenization methods are supported:

Methods

Single-word tokenization

- Name: single_word
- Applies to: all natural languages, such as Chinese, English, and Japanese
- Parameter:
  - caseSensitive: specifies whether this method is case-sensitive. The default value is false. False indicates that all English letters are converted to lowercase letters.
  - delimitWord: specifies whether to tokenize alphanumeric characters. The default value is false.

English letters or numbers are tokenized based on spaces or punctuation, and English letters are converted to lowercase letters. For example, "Hang Zhou" is tokenized into "hang" and "zhou". You can use MatchQuery or MatchPhraseQuery to query data that contains "hang", "HANG", or "Hang". If you do not need the system to automatically convert English letters to lowercase letters, you can set the caseSensitive parameter to true.

Alphanumeric characters such as product models cannot be tokenized by this method because there are no spaces or punctuation between letters and numbers. For example, "IPhone6" remains "IPhone6" after tokenization. When querying by MatchQuery or MatchPhraseQuery, you can retrieve data only by specifying "iphone6 " to query. You can set the delimitWord parameter to true to separate English letters from numbers. This way, "iphone6" is tokenzied into "iphone" and "6".

Delimiter tokenization

- Name: split
- Applies to: all natural languages, such as Chinese, English, and Japanese
- Parameter:
  - delimiter: The default delimiter is a space. You can set the delimiter to any character based on your needs.

Search index tokenizes words based on general dictionaries, but words from some special industries need to be tokenized based on their custom dictionaries. In this case, tokenization methods provided by search index cannot meet the needs of users.

Delimiter tokenization, or custom tokenization, can address this need. Users segment words in their own way and tokenize the segmented words with a specific delimiter. Then, the tokenized words are written to Table Store.

Note:
When you create a search index, the delimiter set in the field for tokenization must be the same as that in the written data. Otherwise, data may not be retrieved.

Minimum semantic unit-based tokenization

- Name: min_word
- Applies to: Chinese

- Parameter:
  - None

In addition to word-level tokenization, search index also provides semantic-level tokenization. By using this method, text is tokenized into minimum semantic units.

In most cases, this method can meet basic requirements in the full-text search scenario.

Maximum semantic unit-based tokenization

- Name: max_word
- Applies to: Chinese
- Parameter:
  - None

Aside from the minimum semantic unit-based tokenization, the more complex maximum semantic unit-based tokenization is provided to obtain as many semantic units as possible. However, different semantic units may overlap. The total length of the tokenized words is greater than the length of the original text. The index fields are increased.

This method can generate more tokens and increase the probability of obtaining results. However, the index fields are greatly increased. MatchPhraseQuery also tokenizes words in the same way. This way, tokens may overlap and data may not be retrieved. Therefore, This tokenization method is more suitable for MatchQuery.

Fuzzy tokenization

- Name: fuzzy
- Applies to: all natural languages, such as Chinese, English, and Japanese
- Parameter:
  - minChars: specifies the minimum number of characters for a token. We recommend that you set this value to 2.
  - maxChars: specifies the maximum number of characters for a token. We recommend that you set this value to a number smaller than or equal to 7.

- Limits:
  - A text field cannot exceed 32 characters in length. Only the first 32 characters of a text field is retained and the characters after the 32nd character are truncated and discarded.

Assume that you need to be able to quickly obtain results for short text, such as headlines, movie names, or book titles by using drop-down prompts. In this case, you can use fuzzy tokenization to tokenize text content into n-grams, whose lengths are between minChars and maxChars.

This method has minimal delay when obtaining results, but the index fields are increased greatly. Therefore, this tokenization method is suitable for short text.

Comparison

The following table compares the five tokenization methods.

	Single-word tokenization	Delimiter tokenization	Minimum semantic unit-based tokenization	Maximum semantic unit-based tokenization	Fuzzy tokenization
Index expansion	Medium	Small	Small	Large	Huge
Relevance	Weak	Weak	Medium	Relatively strong	Relatively strong
Applicable language	All	All	Chinese	Chinese	All
Length limit	No	No	No	No	32 characters
Recall rate	High	Low	Low	Medium	Medium

6.3.9 MatchAllQuery

You can use MatchAllQuery to query the total number of rows or any number of rows in a table.

Example

```
/**
 *  Use   MatchAllQu  ery   to   query   the   total   number   of
 rows   in   a   table .
 * @ param   client
 */
```

```
private   static   void   matchAllQu  ery ( SyncClient   client ) {
     SearchQuer  y   searchQuer  y  =  new   SearchQuer  y ();

    /**
    *  Set   the   query   type   to   MatchAllQu  ery .
    */
     searchQuer  y . setQuery ( new   MatchAllQu  ery ());

    /**
     *  In   the   MatchAllQu  ery - based   query   result ,  the
value   of   TotalCount   is   the   total   number   of   rows   in
  a   table .  This   value   is   an   approximat  e   value   when
  you   query   a   table   that   contains   a   large   number   of
  rows .
     * To   return   only   the   total   number   of   rows   without
  any   specific   data ,  you   can   set   Limit   to   0 .  Then
,  Table   Store   returns   no   data   in   the   rows .
     */
     searchQuer  y . setLimit ( 0 );
     SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );

    /**
    *  Set   the   total   number   of   matched   rows .
    */
     searchQuer  y . setGetTota  lCount ( true );
     SearchResp  onse   resp  =  client . search ( searchRequ  est );
    /**
     *  Check   whether   Table   Store   returns   matched   data
  from   all   partitions .  When   the   value   of   isAllSucce
ss   is   false ,  Table   Store   may   fail   to   query   some
partitions   and   return   a   part   of   data .
     */
     if  (!  resp . isAllSucce  ss ()) {
         System . out . println (" NotAllSucc  ess !") ;
    }
     System . out . println (" IsAllSucce  ss : " +  resp .
isAllSucce  ss ());
     System . out . println (" TotalCount : " +  resp . getTotalCo
unt ()); //  The   total   number   of   rows .
     System . out . println ( resp . getRequest  Id ());
}
```

6.3.10 MatchQuery

You can use MatchQuery to query data in the fields of Text type in full-text search scenarios. Table Store tokenizes the value of Text type in the index and the target value that you specify for the MatchQuery type based on your configuration. Therefore, Table Store can match tokenized terms in a query.

For example, the title field value in a row is "Hangzhou West Lake Scenic Area". Table Store tokenizes the value into "Hangzhou", "West", "Lake", "Scenic", and "Area". If you specify the target term as "Lake Scenic" in MatchQuery, Table Store returns this row in the query result.

Parameters

- fieldName: the name of the target field.
- text: the target term. Table Store tokenizes this term into multiple terms.
- minimumShouldMatch: the minimum number of terms that the value of the fieldName field in a row contains when Table Store returns this row in the query result.
- operator: the operator used in a logical operation. The default operator OR specifies that Table Store returns the row when some of the tokens of the field value in the row match the target term. The operator AND specifies that Table Store returns the row only when all tokens of the field value in the row match the target term.

Example

```
/**
 * Search the table for rows where the value of
 Col_Keywor d matches " hangzhou ". Table Store returns
 matched rows and the total number of matched rows .
 * @ param client
 */
 private static void matchQuery ( SyncClient client ) {
     SearchQuer y searchQuer y = new SearchQuer y ();
     MatchQuery matchQuery = new MatchQuery (); // Set the
 query type to MatchQuery .
     matchQuery . setFieldNa me (" Col_Keywor d "); // Set the
 name of the field that you want to match .
     matchQuery . setText (" hangzhou "); // Set the value that
   you want to match .
     searchQuer y . setQuery ( matchQuery );
     searchQuer y . setOffset ( 0 ); // Set Offset to 0 .
     searchQuer y . setLimit ( 20 ); // Set Limit to 20 to
 return 20 rows or fewer .
     SearchRequ est searchRequ est = new SearchRequ est (
 TABLE_NAME , INDEX_NAME , searchQuer y );
     SearchResp onse resp = client . search ( searchRequ est );
     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ());
     System . out . println (" Row : " + resp . getRows ()); // If
   you do not set columnsToG et , Table Store only
 returns primary keys by default .

     SearchRequ est . ColumnsToG et columnsToG et = new
 SearchRequ est . ColumnsToG et ();
     columnsToG et . setReturnA ll ( true ); // Set columnsToG
 et to true to return all columns .
     searchRequ est . setColumns ToGet ( columnsToG et );

     resp = client . search ( searchRequ est );
     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ()); // The total number of matched rows instead
 of the number of returned rows .
     System . out . println (" Row : " + resp . getRows ());
```

```
}
```

6.3.11 MatchPhraseQuery

This query is similar to MatchQuery, but evaluates the positional relationship between multiple tokens. Table Store exactly matches the order and position of these tokens in the target row.

For example, the field value is "Hangzhou West Lake Scenic Area". If you specify the target term as "Hangzhou Scenic Area" in Query, Table Store returns the row that contains this target term when you use MatchQuery. However, when you use MatchPhraseQuery, Table Store does not return the row that contains this target term . The distance between "Hangzhou" and "Scenic Area" in Query is 0. But the distance in the field is 2, because the two words "West" and "Lake" exist between "Hangzhou" and "Scenic Area".

Parameters

- fieldName: the name of the target field.
- text: the target term. Table Store tokenizes this term into multiple terms before the query.

Example

```
/**
 * Search the table for rows where the value of
 Col_Text matches " hangzhou shanghai ." Table Store returns
   the total number of rows that match the phrase as
   a whole and matched rows in this query .
 * @ param client
 */
 private static void matchPhras eQuery ( SyncClient client )
 {
     SearchQuer y searchQuer y = new SearchQuer y ();
     MatchPhras eQuery matchPhras eQuery = new MatchPhras
 eQuery (); // Set the query type to MatchPhras eQuery .
     matchPhras eQuery . setFieldNa me (" Col_Text "); // Set the
   field that you want to match .
     matchPhras eQuery . setText (" hangzhou shanghai "); // Set
 the value that you want to match .
     searchQuer y . setQuery ( matchPhras eQuery );
     searchQuer y . setOffset ( 0 ); // Set Offset to 0 .
     searchQuer y . setLimit ( 20 ); // Set Limit to 20 to
 return 20 rows or fewer .
     SearchRequ est searchRequ est = new SearchRequ est (
 TABLE_NAME , INDEX_NAME , searchQuer y );
     SearchResp onse resp = client . search ( searchRequ est );
     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ());
     System . out . println (" Row : " + resp . getRows ()); //
 Return primary keys only by default .
```

```
    SearchRequ  est . ColumnsToG  et   columnsToG  et  =  new
SearchRequ  est . ColumnsToG  et ();
    columnsToG  et . setReturnA  ll ( true ); //  Set   columnsToG
et   to   true   to   return   all   columns .
    searchRequ  est . setColumns  ToGet ( columnsToG  et );

    resp  =  client . search ( searchRequ  est );
    System . out . println (" TotalCount : " +  resp . getTotalCo
unt ()); //  The   total   number   of   matched   rows   instead
of   the   number   of   returned   rows .
    System . out . println (" Row : " +  resp . getRows ());
}
```

6.3.12 TermQuery

You can use TermQuery to query data that exactly matches the specified value of a field. When a table contains a Text string, Table Store tokenizes the string and exactly matches any of the tokens. For example, Table Store tokenizes Text string "tablestore is cool" into "tablestore," "is," and "cool". When you specify any of these tokens as a query string, you can retrieve the query result that contains the token.

Parameters

- fieldName: the name of the target field.
- term: the target term. Table Store does not tokenize this term, but exactly matches the whole term.

Example

```
/**
 *  Search   the   table   for   rows   where   the   value   of
 Col_Keywor  d   exactly   matches  " hangzhou ".
 * @ param   client
 */
 private   static   void   termQuery ( SyncClient   client ) {
    SearchQuer  y   searchQuer  y  =  new   SearchQuer  y ();
    TermQuery   termQuery  =  new   TermQuery (); //  Set   the
 query   type   to   TermQuery .
    termQuery . setFieldNa  me (" Col_Keywor  d "); //  Set   the
 name   of   the   field   that   you   want   to   match .
    termQuery . setTerm ( ColumnValu  e . fromString (" hangzhou
 ")); //  Set   the   value   that   you   want   to   match .
    searchQuer  y . setQuery ( termQuery );
    SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
 TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );

    SearchRequ  est . ColumnsToG  et   columnsToG  et  =  new
 SearchRequ  est . ColumnsToG  et ();
    columnsToG  et . setReturnA  ll ( true ); //  Set   columnsToG
 et   to   true   to   return   all   columns .
    searchRequ  est . setColumns  ToGet ( columnsToG  et );

    SearchResp  onse   resp  =  client . search ( searchRequ  est );
    System . out . println (" TotalCount : " +  resp . getTotalCo
 unt ()); //  The   total   number   of   matched   rows   instead
 of   the   number   of   returned   rows .
```

```
    System . out . println (" Row : " +  resp . getRows ());
}
```

6.3.13 TermsQuery

This query is similar to TermQuery, but supports multiple terms. This query is also similar to the SQL IN operator.

Parameters

fieldName: the name of the target field.

terms: the target terms. Table Store returns the data in a row when the system matches one term in the row.

Example

```
/**
 *  Search   the   table   for   rows   where   the   value   of
 Col_Keywor  d   exactly   matches  " hangzhou "  or  " xi ' an ".
 * @ param   client
 */
 private   static   void   termQuery ( SyncClient   client ) {
     SearchQuer  y   searchQuer  y  =  new   SearchQuer  y ();
     TermsQuery   termsQuery  =  new   TermsQuery (); //  Set   the
 query   type   to   TermsQuery .
     termsQuery . setFieldNa  me (" Col_Keywor  d "); //  Set   the
 name   of   the   field   that   you   want   to   match .
     termsQuery . addTerm ( ColumnValu  e . fromString (" hangzhou
 ")); //  Set   the   value   that   you   want   to   match .
     termsQuery . addTerm ( ColumnValu  e . fromString (" xi ' an
 ")); //  Set   the   value   that   you   want   to   match .
     searchQuer  y . setQuery ( termsQuery );
     SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
 TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );

     SearchRequ  est . ColumnsToG  et   columnsToG  et  =  new
 SearchRequ  est . ColumnsToG  et ();
     columnsToG  et . setReturnA  ll ( true ); //  Set   columnsToG
 et   to   true   to   return   all   columns .
     searchRequ  est . setColumns  ToGet ( columnsToG  et );

     SearchResp  onse   resp  =  client . search ( searchRequ  est );
     System . out . println (" TotalCount : " +  resp . getTotalCo
 unt ()); //  The   total   number   of   matched   rows   instead
 of   the   number   of   returned   rows .
     System . out . println (" Row : " +  resp . getRows ());
```

```
}
```

6.3.14 PrefixQuery

You can use PrefixQuery to query data that matches a specified prefix. When a table contains a TEXT string, Table Store tokenizes the string and matches any of the tokens with the specified prefix.

Parameters

- fieldName: the name of the target field.
- prefix: the value of the specified prefix.

Example

```
/**
 *  Search   the   table   for   rows   where   the   value   of
 Col_Keywor  d   contains   the   prefix   that   exactly   matches   "
 hangzhou ".
 * @ param   client
 */
 private   static   void   prefixQuer  y ( SyncClient   client ) {
     SearchQuer  y   searchQuer  y  =  new   SearchQuer  y ();
     PrefixQuer  y   prefixQuer  y  =  new   PrefixQuer  y (); //  Set
   the   query   type   to   PrefixQuer  y .
     prefixQuer  y . setFieldNa  me (" Col_Keywor  d ");
     prefixQuer  y . setPrefix (" hangzhou ");
     searchQuer  y . setQuery ( prefixQuer  y );
     SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
 TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );

     SearchRequ  est . ColumnsToG  et   columnsToG  et  =  new
 SearchRequ  est . ColumnsToG  et ();
     columnsToG  et . setReturnA  ll ( true ); //  Set   columnsToG
 et   to   true   to   return   all   columns .
     searchRequ  est . setColumns  ToGet ( columnsToG  et );

     SearchResp  onse   resp  =  client . search ( searchRequ  est );
     System . out . println (" TotalCount : " +  resp . getTotalCo
 unt ()); //  The   total   number   of   matched   rows   instead
 of   the   number   of   returned   rows .
     System . out . println (" Row : " +  resp . getRows ());
```

6.3.15 RangeQuery

You can use RangeQuery to query data that falls within a specified range. When a table contains a TEXT string, Table Store tokenizes the string and matches any of the tokens that falls within the specified range.

Parameters

- fieldName: the name of the target field.
- from: the value of the start position.

- to: the value of the end position.
- includeLow: specifies whether the query result includes the value of the from parameter. This is a parameter of Boolean type.
- includeUpper: specifies whether the query result includes the value of the to parameter. This is a parameter of Boolean type.

Example

```
/**
 * Search the table for rows where the value of
 Col_Long is greater than 3 . Table Store sorts these
 rows by Col_Long in descending order .
 * @ param client
 */
 private static void rangeQuery ( SyncClient client ) {
     SearchQuer y searchQuer y = new SearchQuer y ();
     RangeQuery rangeQuery = new RangeQuery (); // Set the
 query type to RangeQuery .
     rangeQuery . setFieldNa me (" Col_Long "); // Set the name
   of the target field .
     rangeQuery . greaterTha n ( ColumnValu e . fromLong ( 3 ));
  // Specify the range of the value of the field .
 The required value is larger than 3 .
     searchQuer y . setQuery ( rangeQuery );

    // Sort the result by Col_Long in descending order
 .
     FieldSort fieldSort = new FieldSort (" Col_Long ");
     fieldSort . setOrder ( SortOrder . DESC );
     searchQuer y . setSort ( new Sort ( Arrays . asList (( Sort .
 Sorter ) fieldSort )));

     SearchRequ est searchRequ est = new SearchRequ est (
 TABLE_NAME , INDEX_NAME , searchQuer y );

     SearchResp onse resp = client . search ( searchRequ est );
     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ()); // The total number of matched rows instead
 of the number of returned rows .
     System . out . println (" Row : " + resp . getRows ());

    /**
     * You can specify a value for SearchAfte r to
   start a new query . For example , you can set
 SearchAfte r to 5 and sort the result by Col_Long
 in descending order . Then , you retrieve the rows that
   follow the row whose Col_Long is equal to 5 . This
   is similar to the method where you specify that
 the value of Col_Long is smaller than 5 .
     */
     searchQuer y . setSearchA fter ( new SearchAfte r ( Arrays .
 asList ( ColumnValu e . fromLong ( 5 ))));
     searchRequ est = new SearchRequ est ( TABLE_NAME ,
 INDEX_NAME , searchQuer y );
     resp = client . search ( searchRequ est );
     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ()); // The total number of matched rows instead
 of the number of returned rows .
     System . out . println (" Row : " + resp . getRows ());
```

```
}
```

6.3.16 WildcardQuery

You can use WildcardQuery to query data that matches wildcard characters. You can specify a value you want to match as a string that consists of one or more wildcard characters. An asterisk (*) is interpreted as a number of characters or an empty string. A question mark (?) is interpreted as any single character. For example, when you search the string "table*e", you can retrieve query results such as "tablestore".

Parameters

- fieldName: the name of the target field.
- value: the value that contains one or more wildcard characters. Table Store supports two types of wildcard characters: asterisk (*) and question mark (?). The value cannot start with an asterisk (*) and the length of the value can be 10 bytes or less.

Example

```
/**
 * Search the table for rows where the value of
 Col_Keywor d matches " hang * u ".
 * @ param client
 */
 private static void wildcardQu ery ( SyncClient client ) {
     SearchQuer y searchQuer y = new SearchQuer y ();
     WildcardQu ery wildcardQu ery = new WildcardQu ery
 (); // Set the query type to WildcardQu ery .
     wildcardQu ery . setFieldNa me (" Col_Keywor d ");
     wildcardQu ery . setValue (" hang * u "); // Specify a
 string that contains one or more wildcard characters
 in wildcardQu ery .
     searchQuer y . setQuery ( wildcardQu ery );
     SearchRequ est searchRequ est = new SearchRequ est (
 TABLE_NAME , INDEX_NAME , searchQuer y );

     SearchRequ est . ColumnsToG et columnsToG et = new
 SearchRequ est . ColumnsToG et ();
     columnsToG et . setReturnA ll ( true ); // Set columnsToG
 et to true to return all columns .
     searchRequ est . setColumns ToGet ( columnsToG et );

     SearchResp onse resp = client . search ( searchRequ est );

     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ()); // The total number of matched rows instead
 of the number of returned rows .
     System . out . println (" Row : " + resp . getRows ());
```

```
}
```

6.3.17 BoolQuery

You can use BoolQuery to query data based on a combination of filtering conditions. This query contains one or more subqueries as filtering conditions. Table Store returns the rows that match the subqueries.

You can combine these subqueries in different ways. If you specify these subqueries as mustQueries, Table Store returns the result that matches all these subqueries. If you specify these subqueries as mustNotQueries, Table Store returns the result that matches none of these subqueries.

Parameter

- mustQueries: specifies the subqueries that the query result must match. This parameter is equivalent to the AND operator.
- mustNotQueries: specifies the subqueries that the query result must not match. This parameter is equivalent to the NOT operator.
- shouldQueries: specifies the subqueries that the query result may or may not match. If the query result matches the subqueries, the overall relevance score is higher. This parameter is equivalent to the OR operator.
- minimumShouldMatch: specifies the minimum number of shouldQueries that the query result must match.

Examples

```
/**
 * Use  BoolQuery  to  query  data  that  matches  a
 combinatio n  of  filtering  conditions .
 * @ param  client
 */
 public  static  void  boolQuery ( SyncClient  client ) {
     /**
      * Condition  1 : Use  RangeQuery  to  query  data  where
    the  value  of  Col_Long  is  greater  than  3 .
      */
      RangeQuery  rangeQuery  =  new  RangeQuery ();
      rangeQuery . setFieldNa me (" Col_Long ");
      rangeQuery . greaterTha n ( ColumnValu e . fromLong ( 3 ));

     /**
      * Condition  2 : Use  MatchQuery  to  query  data  where
    the  value  of  Col_Keywor d  matches  " hangzhou ".
      */
      MatchQuery  matchQuery  =  new  MatchQuery (); //  Set  the
 query  type  to  MatchQuery .
      matchQuery . setFieldNa me (" Col_Keywor d "); //  Set  the
 name  of  the  field  that  you  want  to  match .
```

```
     matchQuery . setText (" hangzhou "); //  Set   the   value   that
   you   want   to   match .

     SearchQuer  y   searchQuer  y  =  new   SearchQuer  y ();
    {
        /**
         *  Create   a   query   of   BoolQuery   type   where   the
 result   meets   Conditions   1   and   2   at   the   same   time .
         */
         BoolQuery   boolQuery  =  new   BoolQuery ();
         boolQuery . setMustQue  ries ( Arrays . asList ( rangeQuery
 ,  matchQuery ));
         searchQuer  y . setQuery ( boolQuery );
         SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
 TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );
         SearchResp  onse   resp  =  client . search ( searchRequ  est
 );
         System . out . println (" TotalCount : " +  resp .
 getTotalCo  unt ()); //  Display   the   total   number   of
 matched   rows   instead   of   the   number   of   returned   rows .
         System . out . println (" Row : " +  resp . getRows ());
    }

    {
        /**
         *  Create   a   query   of   BoolQuery   type   where   the
 result   meets   at   least   one   of   Condition   1   and   2 .
         */
         BoolQuery   boolQuery  =  new   BoolQuery ();
         boolQuery . setShouldQ  ueries ( Arrays . asList ( rangeQuery
 ,  matchQuery ));
         boolQuery . setMinimum  ShouldMatc  h ( 1 ); //  Specify
   that   the   result   meets   at   least   one   of   the
 conditions .
         searchQuer  y . setQuery ( boolQuery );
         SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
 TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );
         SearchResp  onse   resp  =  client . search ( searchRequ  est
 );
         System . out . println (" TotalCount : " +  resp .
 getTotalCo  unt ()); //  Display   the   total   number   of
 matched   rows   instead   of   the   number   of   returned   rows .
         System . out . println (" Row : " +  resp . getRows ());
    }
}
```

6.3.18 GeoDistanceQuery

You can use GeoDistanceQuery to query data that falls within a distance from a central point. You can specify the central point and the distance from this central point in the query. Table Store returns the rows where the value of a field falls within the distance from the central point.

Parameters

- fieldName: the name of the target field.
- centerPoint: the central coordinate point that consists of latitude and longitude values.

- distanceInMeter: the distance from the central point. This is a value of Double type . Unit: meters.

Example

```
/**
 *  Search   the   table   for   rows   where   the   value   of
 Col_GeoPoi  nt   falls   within   a   specified   distance   from   a
   specified   central   point .
 * @ param   client
 */
 public   static   void   geoDistanc  eQuery ( SyncClient   client ) {
     SearchQuer  y   searchQuer  y  =  new   SearchQuer  y ();
     GeoDistanc  eQuery   geoDistanc  eQuery  =  new   GeoDistanc
 eQuery ();  //  Set   the   query   type   to   GeoDistanc  eQuery .
     geoDistanc  eQuery . setFieldNa  me (" Col_GeoPoi  nt ");
     geoDistanc  eQuery . setCenterP  oint (" 5 , 5 "); //  Specify
 coordinate  s   for   a   central   point .
     geoDistanc  eQuery . setDistanc  eInMeter ( 10000 ); //  You
 can   specify   10 , 000   meters   or   less   as   the   distance
 from   the   central   point .
     searchQuer  y . setQuery ( geoDistanc  eQuery );

     SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
 TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );

     SearchRequ  est . ColumnsToG  et   columnsToG  et  =  new
 SearchRequ  est . ColumnsToG  et ();
     columnsToG  et . setColumns ( Arrays . asList (" Col_GeoPoi  nt
 "));  // Specify   Col_GeoPoi  nt   as   the   column   that   you
 want   to   return .
     searchRequ  est . setColumns  ToGet ( columnsToG  et );

     SearchResp  onse   resp  =  client . search ( searchRequ  est );
     System . out . println (" TotalCount : " +  resp . getTotalCo
 unt ()); //  The   total   number   of   matched   rows   instead
 of   the   number   of   returned   rows .
     System . out . println (" Row : " +  resp . getRows ());
}
```

6.3.19 GeoBoundingBoxQuery

You can use GeoBoundingBoxQuery to query data that falls within a geographic rectangular area. You can specify the geographic rectangular area as a filtering condition in the query. Table Store returns the rows where the value of a field falls within the geographic rectangular area.

Parameters

- fieldName: the name of the target field.
- topLeft: coordinates of the upper-left corner of the geographic rectangular area.
- bottomRight: coordinates in the lower-right corner of the geographic rectangula r area. You can use the upper-left corner and lower-right corner to determine a unique geographic rectangular area.

Example

```
/**
 * The data type of Col_GeoPoi nt is Geopoint . You
 can obtain the rows where the value of Col_GeoPoi nt
  falls within a geographic rectangula r area . For the
  geographic rectangula r area , the upper - left vertex
 is " 10 , 0 " and the lower - right vertex is " 0 , 10 ".
 * @ param client
 */
 public static void geoBoundin gBoxQuery ( SyncClient client
 ) {
     SearchQuer y searchQuer y = new SearchQuer y ();
     GeoBoundin gBoxQuery geoBoundin gBoxQuery = new
 GeoBoundin gBoxQuery (); // Set the query type to
 GeoBoundin gBoxQuery .
     geoBoundin gBoxQuery . setFieldNa me (" Col_GeoPoi nt "); //
 Set the name of the field that you want to match
 .
     geoBoundin gBoxQuery . setTopLeft (" 10 , 0 "); // Specify
  coordinate s for the upper - left vertex of the
 geographic rectangula r area .
     geoBoundin gBoxQuery . setBottomR ight (" 0 , 10 "); //
 Specify coordinate s for the lower - right vertex of
 the geographic rectangula r area .
     searchQuer y . setQuery ( geoBoundin gBoxQuery );

     SearchRequ est searchRequ est = new SearchRequ est (
 TABLE_NAME , INDEX_NAME , searchQuer y );

     SearchRequ est . ColumnsToG et columnsToG et = new
 SearchRequ est . ColumnsToG et ();
     columnsToG et . setColumns ( Arrays . asList (" Col_GeoPoi nt
 ")); // Specify Col_GeoPoi nt as the column that you
 want to return .
     searchRequ est . setColumns ToGet ( columnsToG et );

     SearchResp onse resp = client . search ( searchRequ est );
     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ()); // The total number of matched rows instead
 of the number of returned rows .
     System . out . println (" Row : " + resp . getRows ());
}
```

6.3.20 GeoPolygonQuery

You can use GeoPolygonQuery to query data that falls within a geographic polygon area. You can specify the geographic polygon area as a filtering condition in the query. Table Store returns the rows where the value of a field falls within the geographic polygon area.

Parameters

- fieldName: the name of the target field.
- points: the coordinate points that compose the geographic polygon.

Example

```
/**
 * Search the table for rows where the value of
   Col_GeoPoi nt falls within a specified geographic
 polygon area .
 * @ param client
 */
 public static void geoPolygon Query ( SyncClient client ) {
     SearchQuer y searchQuer y = new SearchQuer y ();
     GeoPolygon Query geoPolygon Query = new GeoPolygon Query
 (); // Set the query type to GeoPolygon Query .
     geoPolygon Query . setFieldNa me (" Col_GeoPoi nt ");
     geoPolygon Query . setPoints ( Arrays . asList (" 0 , 0 "," 5 ,
 5 "," 5 , 0 ")); // Specify coordinate s for vertices of
 the geographic polygon .
     searchQuer y . setQuery ( geoPolygon Query );

     SearchRequ est searchRequ est = new SearchRequ est (
 TABLE_NAME , INDEX_NAME , searchQuer y );

     SearchRequ est . ColumnsToG et columnsToG et = new
 SearchRequ est . ColumnsToG et ();
     columnsToG et . setColumns ( Arrays . asList (" Col_GeoPoi nt
 ")); // Specify Col_GeoPoi nt as the column that you
 want to return .
     searchRequ est . setColumns ToGet ( columnsToG et );

     SearchResp onse resp = client . search ( searchRequ est );
     System . out . println (" TotalCount : " + resp . getTotalCo
 unt ()); // The total number of matched rows instead
 of the number of returned rows .
     System . out . println (" Row : " + resp . getRows ());
}
```

6.3.21 ExistQuery

ExistQuery is also called a null query. It is usually used in queries for sparse data to determine whether a column of a row has a value. For example, ExistQuery is used to query the rows in which the value of the address column is not null.

Note:

If you want to query whether a column contains null values, you must use ExistQuery and the bool query with the must_not clause.

Parameter

fieldName: the column name

Examples

```
/**
 * Use ExistQuery to query the rows in which the
 value of the address column is not null .
 * @ param client
 */
```

```
 private   static   void   termsQuery ( SyncClient   client ) {
     SearchQuer  y   searchQuer  y  =  new   SearchQuer  y ();
     ExistsQuer  y   existQuery  =  new   ExistsQuer  y (); //  Set
 the   query   type   to   ExistsQuer  y .
     existQuery . setFieldNa  me (" address ");
     searchQuer  y . setQuery ( termsQuery );
     SearchRequ  est   searchRequ  est  =  new   SearchRequ  est (
 TABLE_NAME ,  INDEX_NAME ,  searchQuer  y );

     SearchRequ  est . ColumnsToG  et   columnsToG  et  =  new
 SearchRequ  est . ColumnsToG  et ();
     columnsToG  et . setReturnA  ll ( true ); //  Set   ReturnAll
 to   true   to   return   all   columns .
     searchRequ  est . setColumns  ToGet ( columnsToG  et );

     SearchResp  onse   resp  =  client . search ( searchRequ  est );

     System . out . println (" TotalCount : " +  resp . getTotalCo
 unt ()); //  Display   the   total   number   of   matched   rows
 instead   of   the   number   of   returned   rows .
     System . out . println (" Row : " +  resp . getRows ());
}
```

6.4 Limits

This topic describes the limits on using search index.

Mapping

Item	Maximum value	Description
Index fields	200	The number of fields that can be indexed.
EnableSortAndAgg fields	100	The number of fields that can be sorted and aggregated.
Nested levels	1	The number of nested levels.
Nested fields	25	The number of nested fields.
Total length of primary key columns	1,000	The total length of all primary key columns is up to 1,000 Bytes.
Total length of strings in primary key columns	1,000	The total length of strings in all primary key columns is up to 1,000 Bytes.
String length in each attribute column (keyword index)	4 KB	None

Item	Maximum value	Description
String length in each attribute column (text index)	2 MB	Same as the length limit of attribute columns in tables.
String length of a query that contains wildcards	10	The string length of a query that contains wildcards is up to 10 characters.

Search

Item	Maximum value	Description
offset + limit	2,000	To read more than 2000 rows, you must specify the next_token parameter.
timeout	10s	-
Capacity unit (CU)	100,000	· Scanning and analysis requests do not apply. · If your business requirement exceeds this limit, submit a ticket.
QPS	100,000	· The upper limit for lightweight transaction processing is 100,000 queries per second (QPS). · Each index is allowed to take up to 8-core CPU for analytical queries or text queries because each request takes a long time. · The preceding limits are default. If your business requirement exceeds the default limits, submit a ticket.

Index

Item	Maximum value	Description
Rate	50,000 rows /s	· Table Store requires several minutes for load balancing when writing data to a table for the first time, or when a large amount of data is required to be written in a short span of time. · Text field-based indexing is limited to 10,000 rows per second because this indexing consumes more CPU resources for tokenization. · If your business requirement exceeds this limit, submit a ticket.
Synchronization latency	10s	· The value is less than 10s when the writing rate is steady. · The synchronization latency is within one minute in most cases. · New indexes need to be initialized. This process takes up to one minute.
Number of rows	10 billion	If your business requirement exceeds this limit, submit a ticket.
Total size	10 TB	If your business requirement exceeds this limit, submit a ticket.

Other limits

Item	Value
Applicable regions	China (Beijing), China (Shanghai), China (Hangzhou), China (Shenzhen), Singapore, India (Mumbai), China(Hong Kong), and China (Zhangjiakou-Beijing Winter Olympics)
Regions to be applicable	US (Silicon Valley)

Note:

If your business requirement exceeds the default limits, submit a ticket. Describe the scenario, limit item, requirement, and reason in the ticket. Your requirement will be considered in future developments.

7 Global secondary indexes

7.1 Overview

Before you use the Global Secondary Index structure, you need to understand the following terms, limits, and notes.

Terms

Term	Description
Index	You can create an index for some columns in a primary table. The index is read-only.
Pre-defined column	Table Store uses a schema-free model. You can write the unlimited number of columns to a row. You do not need to specify a fixed number of attributes in a schema. You can also pre-define some columns and specify their data types when you create a table.
Single-column index	You can create an index only for one column.
Composite index	You can create an index for multiple columns in a table. A composite index can have Indexed columns 1 and 2.
Indexed attribute column	You can map pre-defined columns in a primary table to non-primary key columns in an index.
Autocomplete	Table Store automatically adds the primary key column that you have not specified in a primary table to an index when you create the index.

Limits

- You can create a maximum of five indexes in a primary table. You cannot create more indexes if you have created five indexes.

- An index contains a maximum of four indexed columns. These indexed columns include a combination of primary keys and pre-defined columns of the primary table. If you specify more indexed columns, you cannot create the index.
- An index contains a maximum of eight attribute columns. If you specify more attribute columns, you cannot create the index.
- You can specify an indexed column as Integer, String, or Binary type. The constraint of Indexed columns is the same as that for primary keys of the primary table.
- If an index combines multiple columns, the size limit for the index is the same as that for primary keys of the primary table.
- When you specify the column of String or Binary type as an attribute column of an index, the limits for the attribute column are the same as those for the attribute column of the primary table.
- You cannot create an index in a table that has Time To Live (TTL) configured. If you want to index a table that has TTL configured, use DingTalk to request technical support.
- You cannot create an index in a table that has Max Versions configured. If a table has Max Versions configured, you cannot create any index for the table. If you index the table, you cannot use the Max Versions feature.
- You cannot customize versions when writing data to an indexed primary table. Otherwise, you cannot write data to the primary table.
- You cannot use the Stream feature in an index.
- An indexed primary table cannot contain repeated rows that have the same primary key during the same BatchWrite operation. Otherwise, you cannot write data to the primary table.

Notes

- Table Store automatically adds the primary key column that you have not specified to the index. When you scan an index, you must specify the range of primary key columns. The range can be from negative infinity to positive infinity. For example, a primary table includes Primary keys `PK0` and `PK1` and Pre-defined column `Defined0`.

  When you create an index for the `Defined0` column, Table Store generates an index that has Primary keys `Defined0`, `PK0`, and `PK1`. When you create an index for the `Defined0` and `PK1` columns, Table Store generates an index that has Primary keys `Defined0` , `PK1` , and `PK0` . When you create an index for the `primary key` columns, Table Store generates an index that has Primary keys `PK1` and `PK0` . When you create an index, you can only specify the column that you want to index. Table Store automatically adds the target columns to the index. For example, a primary table contains Primary keys PK0 and PK1 and Pre-defined column Defined0.

- When you create an index for the Defined0 column, Table Store generates the index that has Primary keys Defined0, PK0, and PK1.
- When you create an index for the PK1 column, Table Store generates the index that has Primary keys PK1 and PK0.

- You can specify pre-defined columns as attribute columns in the primary table. When you specify a pre-defined attribute as an indexed attribute column, you can search this index for the attribute value instead of searching the primary table. However, this increases storage costs. If you do not specify a pre-defined attribute as an indexed attribute column, you have to search the primary table. You can choose between these methods based on your requirements.
- We recommend that you do not specify a column related to the time or date as the first primary key column of an index. This type of column may slow down index updates. We recommend that you hash the column related to the time or date and create an index for the hashed column. To solve related issues, use DingTalk to request technical support.
- We recommend that you do not define an attribute of low cardinality, even an attribute that contains enumerated values, as the first primary key column of an index. For example, the `gender` attribute restricts the horizontal scalability of the index and leads to poor write performance.

7.2 Introduction

A global secondary index in Tablestore has the following features:

- Supports asynchronous data synchronization between a table and table indexes . Under normal network conditions, the data synchronization latency is in milliseconds.

- Supports single-field indexes, compound indexes, and covered indexes. Pre-defined attributes are attributes specified in advance in a table. You can create an index on any pre-defined attribute or on a table primary key. In addition, you can specify a table pre-defined attributes as index attributes or choose not to specify attributes. If you specify pre-defined attributes as the index attributes, you can directly query this index to retrieve data from the base table instead of querying the table. For example, a base table includes three primary keys PK0, PK1, and PK2. Additionally, the table have three pre-defined attributes Defined0, Defined1, and Defined2.
  - You can create an index on PK2 without specifying an attribute.
  - You can create an index on PK2 and specify Defined0 as an attribute.
  - You can create an index on PK3 and PK2 without specifying an attribute.
  - You can create an index on PK3 and PK2 and specify Defined0 as an attribute.
  - You can create an index on PK2, PK1, and PK3 and specify Defined0, Defined1, and Defined2 as an attribute.
  - You can create an index on Defined0 without specifying an attribute.
  - You can create an index on Define0 and PK1 and specify Defined1 as an attribute .
  - You can create an index on Define1 and Define0 without specifying an attribute.
  - You can create an index on Define1 and Define0 and specify Defined2 as an attribute.
- Supports sparse indexes. You can specify a base table pre-defined attribute as an index attribute. This row will be indexed even when all primary keys exist despite the pre-defined attribute being excluded from the base table row. However, this row will not be indexed when a row excludes one or more indexed attributes. For example, a base table includes three primary keys that are PK0, PK1, and PK2. Additionally, the table have three pre-defined attributes Defined0, Defined1, and Defined2. You can create an index on Defined0 and Defined1, and specify Defined2 as an attribute.
  - An index will include a row in a base table that excludes the Defined2 attribute and includes pre-defined attributes Defined0 and Defined1.
  - This row is excluded from the index when a base table row excludes Defined1 but includes the pre-defined attributes Defined0 and Defined2.

- Supports creating and deleting indexes on an existing base table. Existing data in a base table will be copied to an index when you create this index on the base table.
- When you query an index, the query is not automatically performed on the base table of the created index. You need to query the base table. This feature will be supported in later versions.

7.3 Scenarios

The global secondary index is a new Table Store feature. When you create a table, the primary index is composed of all the primary keys. Table Store uses primary keys to uniquelyidentify each row in a table. However, you need to query a table by attributes , primary keys, or primary keys that are not from the first column in more scenarios. Due to insufficient indexes, you can only fetch the results by scanning the entire table and setting filter conditions. If you obtain few results after querying a table with large data volume, the query can cause excessive consumption of resources.

The Table Store Global secondary index feature is similar to that of DynamoDB GSI and HBase Phoenix. You can create an index with one or more specified attributes. In addition, you can sort data in the created index by specified attributes. Every data you write to a base table will be asynchronously synchronized to the created index on the base table. You only have to write data to a base table, and can query indexes created on this base table. This configuration greatly improves query performance in most scenarios. For example, you can create a base table for a common phone log query as follows:

CellNumber	StartTime (Unix timestamps)	CalledNumber	Duration	BaseStationNumber
123456	1532574644	654321	60	1
234567	1532574714	765432	10	1
234567	1532574734	123456	20	3
345678	1532574795	123456	5	2
345678	1532574861	123456	100	2
456789	1532584054	345678	200	3

- `CellNumber` and `StartTime` are primary keys that represent `a calling number` and `the start time of a call`, respectively.
- `CalledNumber`, `Duration`, and `BaseStationNumber` are pre-defined attributes that represent `a called number`, `call duration`, and `the base station number`.

When you end a phone call, the call information is written to this table. You can create global secondary indexes on `CalledNumber` and `BaseStationNumber` respectively to meet various query requirements. For more information about how to create an index, see example in Appendix.

If you have the following query requirements:

- You want to fetch the rows where the CellNumber value matches `234567`.

  You can sort data by primary keys in Table Store. In addition, you can call the `getRange` method to scan data sequentially. When you call the `getRange` method, you need to specify `234567` both as the minimum and maximum values for PK0 (CellNumber). Meanwhile, you need to specify `0` as the minimum value of PK1 (StartTime) and specify `INT_MAX` as the maximum value of PK1. Then you can query the base table.

```
private static void getRangeFromMainTable(SyncClient client, long cellNumber)
{
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);

    // You can specify primary keys.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.fromLong(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong(0));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());

    // You can specify primary keys.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.fromLong(cellNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX);
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());

    rangeRowQueryCriteria.setMaxVersions(1);
```

```
    String   strNum  =  String . format ("% d ",  cellNumber );
    System . out . println (" A   cell   number  " +  strNum  + "
 makes   the   following   calls :");
    while  ( true ) {
        GetRangeRe  sponse   getRangeRe  sponse  =  client .
 getRange ( new   GetRangeRe  quest ( rangeRowQu  eryCriteri  a ));
        for  ( Row   row  :  getRangeRe  sponse . getRows ()) {
            System . out . println ( row );
        }

        //  If   the   value   of   nextStartP  rimaryKey   is   not
  null , you   can   continue   to   read   data   from   the
 base   table .
        if  ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey () !
 =  null ) {
            rangeRowQu  eryCriteri  a . setInclusi  veStartPri
 maryKey ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey ());
        } else  {
            break ;
        }
    }
}
```

- If you want to fetch the rows where the value of CalledNumber is `123456`.

Table Store sorts all rows by primary keys. Because CalledNumber is a pre-defined attribute, you cannot directly query a table by this attribute. Therefore, you can query an index that is created on `CalledNumb  er`.

`IndexOnBeC  alledNumbe  r`:

PK0	PK1	PK2
CalledNumber	CellNumber	StartTime
123456	234567	1532574734
123456	345678	1532574795
123456	345678	1532574861
654321	123456	1532574644
765432	234567	1532574714
345678	456789	1532584054

Note:

Table Store will auto complement primary keys of an index. When building this index, Table Store adds all primary keys of a base table to an index created on this base table. Therefore, the index includes three primary keys.

Because `IndexOnBeC alledNumbe r` is an index that is created on CalledNumber, you can directly query this index to fetch results.

```
private  static  void  getRangeFr omIndexTab le ( SyncClient
client ,  long  cellNumber ) {
    RangeRowQu eryCriteri a  rangeRowQu eryCriteri a  =  new
RangeRowQu eryCriteri a ( INDEX0_NAM E );

   //  You  can  specify  primary  keys .
    PrimaryKey Builder  startPrima ryKeyBuild er  =
PrimaryKey Builder . createPrim aryKeyBuil der ();
    startPrima ryKeyBuild er . addPrimary KeyColumn (
DEFINED_CO L_NAME_1 ,  PrimaryKey Value . fromLong ( cellNumber
));
    startPrima ryKeyBuild er . addPrimary KeyColumn (
PRIMARY_KE Y_NAME_1 ,  PrimaryKey Value . INF_MIN );
    startPrima ryKeyBuild er . addPrimary KeyColumn (
PRIMARY_KE Y_NAME_2 ,  PrimaryKey Value . INF_MIN );
    rangeRowQu eryCriteri a . setInclusi veStartPri maryKey (
startPrima ryKeyBuild er . build ());

   //  You  can  specify  primary  keys .
    PrimaryKey Builder  endPrimary KeyBuilder  =  PrimaryKey
Builder . createPrim aryKeyBuil der ();
    endPrimary KeyBuilder . addPrimary KeyColumn ( DEFINED_CO
L_NAME_1 ,  PrimaryKey Value . fromLong ( cellNumber ));
    endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME_1 ,  PrimaryKey Value . INF_MAX );
    endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME_2 ,  PrimaryKey Value . INF_MAX );
    rangeRowQu eryCriteri a . setExclusi veEndPrima ryKey (
endPrimary KeyBuilder . build ());

    rangeRowQu eryCriteri a . setMaxVers ions ( 1 );

    String  strNum  =  String . format ("% d ",  cellNumber );
    System . out . println (" A  cell  number " +  strNum  + "
was  called  by  the  following  numbers ");
    while  ( true ) {
        GetRangeRe sponse  getRangeRe sponse  =  client .
getRange ( new  GetRangeRe quest ( rangeRowQu eryCriteri a ));
        for  ( Row  row  :  getRangeRe sponse . getRows ()) {
            System . out . println ( row );
       }

       //  If  the  value  of  nextStartP rimaryKey  is  not
  null ,  you  can  continue  to  read  data  from  the
base  table .
        if  ( getRangeRe sponse . getNextSta rtPrimaryK ey () !
=  null ) {
            rangeRowQu eryCriteri a . setInclusi veStartPri
maryKey ( getRangeRe sponse . getNextSta rtPrimaryK ey ());
       }  else  {
            break ;
       }
   }
```

```
}
```

- If you want to fetch the rows where the value of BaseStationNumber matches `002` and the value of StartTime matches `1532574740`.

  This query specifies both `BaseStatio nNumber` and `StartTime` as conditions. Therefore, you can create a compound index on the `BaseStatio nNumber` and `StartTime`.

  `IndexOnBas eStation1`:

PK0	PK1	PK2
BaseStationNumber	StartTime	CellNumber
001	1532574644	123456
001	1532574714	234567
002	1532574795	345678
002	1532574861	345678
003	1532574734	234567
003	1532584054	456789

  You can query the `IndexOnBas eStation1` index.

```
private static void getRangeFr omIndexTab le ( SyncClient
client ,
                                            long baseStatio
nNumber ,
                                            long startTime ) {
    RangeRowQu eryCriteri a rangeRowQu eryCriteri a = new
RangeRowQu eryCriteri a ( INDEX1_NAM E );

   // You can specify primary keys .
    PrimaryKey Builder startPrima ryKeyBuild er =
PrimaryKey Builder . createPrim aryKeyBuil der ();
    startPrima ryKeyBuild er . addPrimary KeyColumn (
DEFINED_CO L_NAME_3 , PrimaryKey Value . fromLong ( baseStatio
nNumber ));
    startPrima ryKeyBuild er . addPrimary KeyColumn (
PRIMARY_KE Y_NAME_2 , PrimaryKey Value . fromLong ( startTime
));
    startPrima ryKeyBuild er . addPrimary KeyColumn (
PRIMARY_KE Y_NAME_1 , PrimaryKey Value . INF_MIN );
    rangeRowQu eryCriteri a . setInclusi veStartPri maryKey (
startPrima ryKeyBuild er . build ());

   // You can specify primary keys .
    PrimaryKey Builder endPrimary KeyBuilder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
    endPrimary KeyBuilder . addPrimary KeyColumn ( DEFINED_CO
L_NAME_3 , PrimaryKey Value . fromLong ( baseStatio nNumber ));
```

```
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_2 ,  PrimaryKey  Value . INF_MAX );
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_1 ,  PrimaryKey  Value . INF_MAX );
    rangeRowQu  eryCriteri  a . setExclusi  veEndPrima  ryKey (
endPrimary  KeyBuilder . build ());

    rangeRowQu  eryCriteri  a . setMaxVers  ions ( 1 );

    String   strBaseSta  tionNum  =  String . format ("% d ",
baseStatio  nNumber );
    String   strStartTi  me  =  String . format ("% d ",  startTime
);
    System . out . println (" All   called   numbers   forwarded
  by   the   base   station " +  strBaseSta  tionNum  + "  that
start   from " +  strStartTi  me  + " are   listed   as   follows
:");
    while  ( true ) {
        GetRangeRe  sponse   getRangeRe  sponse  =  client .
getRange ( new   GetRangeRe  quest ( rangeRowQu  eryCriteri  a ));
        for  ( Row   row   :  getRangeRe  sponse . getRows ()) {
            System . out . println ( row );
        }

        //  If   the   nextStartP  rimaryKey   value   is   not
null , you   can   continue   to   read   data   from   the   base
  table .
        if  ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey () !
=  null ) {
            rangeRowQu  eryCriteri  a . setInclusi  veStartPri
maryKey ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey ());
        }  else  {
            break ;
        }
    }
}
```

- **If you want to fetch the rows where the value of BaseStationNumber `003` matches the StartTime value range from `1532574861` to `1532584054`. Only the Duration will be displayed in the rows.**

  In this query, you specify both `BaseStatio  nNumber` and `StartTime` as conditions. Only `Duration` appears in the result set. You can issue a query on the last index, and then fetch Duration by querying the base table.

```
private   static   void   getRowFrom  IndexAndMa  inTable (
SyncClient   client ,
                                                long   baseStatio
nNumber ,
                                                long   startTime ,
                                                long   endTime ,
                                                String   colName )
{
    RangeRowQu  eryCriteri  a   rangeRowQu  eryCriteri  a  =  new
RangeRowQu  eryCriteri  a ( INDEX1_NAM  E );

   //  You   can   specify   primary   keys .
    PrimaryKey  Builder   startPrima  ryKeyBuild  er  =
PrimaryKey  Builder . createPrim  aryKeyBuil  der ();
```

```
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
DEFINED_CO  L_NAME_3 ,  PrimaryKey  Value . fromLong ( baseStatio
nNumber ));
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_2 ,  PrimaryKey  Value . fromLong ( startTime
));
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_1 ,  PrimaryKey  Value . INF_MIN );
    rangeRowQu  eryCriteri  a . setInclusi  veStartPri  maryKey (
startPrima  ryKeyBuild  er . build ());

   //  You   can   specify   primary   keys .
    PrimaryKey  Builder   endPrimary  KeyBuilder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( DEFINED_CO
L_NAME_3 ,  PrimaryKey  Value . fromLong ( baseStatio  nNumber ));
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_2 ,  PrimaryKey  Value . fromLong ( endTime ));
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_1 ,  PrimaryKey  Value . INF_MAX );
    rangeRowQu  eryCriteri  a . setExclusi  veEndPrima  ryKey (
endPrimary  KeyBuilder . build ());

    rangeRowQu  eryCriteri  a . setMaxVers  ions ( 1 );

    String   strBaseSta  tionNum  =  String . format ("% d ",
baseStatio  nNumber );
    String   strStartTi  me  =  String . format ("% d ",  startTime
);
    String   strEndTime  =  String . format ("% d ",  endTime );

    System . out . println (" The   list   of   calls   forwarded
  by   the   base   station " +  strBaseSta  tionNum  + " from "
+  strStartTi  me  + " to " +  strEndTime  + " is   listed   as
follows :");
    while  ( true ) {
        GetRangeRe  sponse   getRangeRe  sponse  =  client .
getRange ( new   GetRangeRe  quest ( rangeRowQu  eryCriteri  a ));
        For  ( Row   row :  fig .  getrows  ()){
            PrimaryKey   curIndexPr  imaryKey  =  row . getPrimary
Key ();
            PrimaryKey  Column   mainCalled  Number  =  curIndexPr
imaryKey . getPrimary  KeyColumn ( PRIMARY_KE  Y_NAME_1 );
            PrimaryKey  Column   callStartT  ime  =  curIndexPr
imaryKey . getPrimary  KeyColumn ( PRIMARY_KE  Y_NAME_2 );
            PrimaryKey  Builder   mainTableP  KBuilder  =
PrimaryKey  Builder . createPrim  aryKeyBuil  der ();
            mainTableP  KBuilder . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_1 ,  mainCalled  Number . getValue ());
            mainTableP  KBuilder . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_2 ,  callStartT  ime . getValue ());
            PrimaryKey   mainTableP  K  =  mainTableP  KBuilder .
build (); //  You   can   specify   primary   keys   for   the
base   table .

           //  You   can   query   the   base   table .
            SingleRowQ  ueryCriter  ia   criteria  =  new
SingleRowQ  ueryCriter  ia ( TABLE_NAME ,  mainTableP  K );
            criteria . addColumns  ToGet ( colName ); //  You   can
  read   the   Duration   attribute   value   of   the   base
table .
           //  You   can   specify   1   to   indicate   the
latest   data   version   will   be   read .
            criteria . setMaxVers  ions ( 1 );
```

```
            GetRowResp  onse   getRowResp  onse  =  client . getRow
 ( new   GetRowRequ  est ( criteria ));
            Row   mainTableR  ow  =  getRowResp  onse . getRow ();

            System . out . println ( mainTableR  ow );
        }

        //  If   the   nextStartP  rimaryKey   value   is   not
 null ,  you   can   continue   to   read   data   from  the   base
   table .
         if  ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey () !
 =  null ) {
            rangeRowQu  eryCriteri  a . setInclusi  veStartPri
 maryKey ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey ());
        }  else  {
            break ;
        }
    }
}
```

To improve query performance, you can create a compound index on `BaseStatio nNumber` and `StartTime`. You can specify `Duration` as an attribute of this index.

The following index is created.

`IndexOnBas eStation2`:

PK0	PK1	PK2	Defined0
BaseStatio nNumber	StartTime	CellNumber	Duration
001	1532574644	123456	600
001	1532574714	234567	10
002	1532574795	345678	5
002	1532574861	345678	100
003	1532574734	234567	20
003	1532584054	456789	200

You can query the `IndexOnBas eStation2` index:

```
 private   static   void   getRangeFr  omIndexTab  le ( SyncClient
 client ,
                                             long   baseStatio
 nNumber ,
                                             long   startTime ,
                                             long   endTime ,
                                             String   colName ) {
     RangeRowQu  eryCriteri  a   rangeRowQu  eryCriteri  a  =  new
 RangeRowQu  eryCriteri  a ( INDEX2_NAM  E );

    //  You   can   specify   primary   keys .
```

```
    PrimaryKey  Builder   startPrima  ryKeyBuild  er  =
PrimaryKey  Builder . createPrim  aryKeyBuil  der ();
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
DEFINED_CO  L_NAME_3 ,  PrimaryKey  Value . fromLong ( baseStatio
nNumber ));
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_2 ,  PrimaryKey  Value . fromLong ( startTime
));
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_1 ,  PrimaryKey  Value . INF_MIN );
    rangeRowQu  eryCriteri  a . setInclusi  veStartPri  maryKey (
startPrima  ryKeyBuild  er . build ());

   //  You   can   specify   primary   keys .
    PrimaryKey  Builder   endPrimary  KeyBuilder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( DEFINED_CO
L_NAME_3 ,  PrimaryKey  Value . fromLong ( baseStatio  nNumber ));
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_2 ,  PrimaryKey  Value . fromLong ( endTime ));
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_1 ,  PrimaryKey  Value . INF_MAX );
    rangeRowQu  eryCriteri  a . setExclusi  veEndPrima  ryKey (
endPrimary  KeyBuilder . build ());

   //  You   can   specify   the   attribute   name   to   read .
    rangeRowQu  eryCriteri  a . addColumns  ToGet ( colName );

    rangeRowQu  eryCriteri  a . setMaxVers  ions ( 1 );

    String   strBaseSta  tionNum  =  String . format ("% d ",
baseStatio  nNumber );
    String   strStartTi  me  =  String . format ("% d ",  startTime
);
    String   strEndTime  =  String . format ("% d ",  endTime );

    System . out . println (" The   duration   of   calls
forwarded   by   the   base   station " +  strBaseSta  tionNum  + "
from " +  strStartTi  me  + " to " +  strEndTime  + " is   listed
as   follows :");
    while  ( true ) {
        GetRangeRe  sponse   getRangeRe  sponse  =  client .
getRange ( new   GetRangeRe  quest ( rangeRowQu  eryCriteri  a ));
        for  ( Row   row  :  getRangeRe  sponse . getRows ()) {
            System . out . println ( row );
        }

        //  If   the   nextStartP  rimaryKey   value   is   not
null ,  you   can   continue   to   read   data   from   the   base
  table .
        if  ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey () !
=  null ) {
            rangeRowQu  eryCriteri  a . setInclusi  veStartPri
maryKey ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey ());
        }  else  {
            break ;
        }
    }
}
```

```
```

Hence, if you do not specify `Duration` as an index attribute, you have to retrieve Duration by querying the base table. However, when you specify `Duration` as an index attribute, this attribute data is stored in the base table and the index. The configuration improves query performance at the cost of disk space consumption.

- If you want to fetch the following values from a result set: total call duration, the average call duration, the maximum call duration, and the minimum call duration.This result set is a value of BaseStationNumber `003` with a StartTime value range from `1532574861` to `1532584054`.

  Compared to the last query, return is not required for each call duration. However, return is required for duration statistics. You can fetch results using the same method as the last query. Then you can perform Duration calculations to obtain the required result. In addition, you can execute SQL statements in SQL-on-OTS to obtain statistics. For more information about how to activate SQL-on-OTS, see OLAP on Table Store: serverless SQL big data analysis on Data Lake Analytics. You can use most MySQL syntax in SQL-on-OTS. Additionally, with SQL-on-OTS, you can easily process complicated calculations that are applicable to your business.

7.4 Java SDK for global secondary indexes

In this section, you can call the `createTable` method and the `scanFromIndex` method in the Java SDK to perform the following operations.

- You can create a base table and an index on this base table at the same time.

```
private   static   void   createTabl  e ( SyncClient   client ) {
    TableMeta   tableMeta  =  new   TableMeta ( TABLE_NAME );
    tableMeta . addPrimary  KeyColumn ( new   PrimaryKey  Schema (
PRIMARY_KE  Y_NAME_1 ,  PrimaryKey  Type . STRING )); //  You   can
  specify   a   primary   key   for   a   base   table .
    tableMeta . addPrimary  KeyColumn ( new   PrimaryKey  Schema
( PRIMARY_KE  Y_NAME_2 ,  PrimaryKey  Type . INTEGER )); //  Set
primary   key   for  the   base   table
    tableMeta . addDefined  Column ( new   DefinedCol  umnSchema (
DEFINED_CO  L_NAME_1 ,  DefinedCol  umnType . STRING )); //  You
can   specify   a   pre - defined   attribute   for   the   base
table .
    tableMeta . addDefined  Column ( new   DefinedCol  umnSchema (
DEFINED_CO  L_NAME_2 ,  DefinedCol  umnType . INTEGER )); //  You
can   specify   a   pre - defined   attribute   for   the   base
table .
    tableMeta . addDefined  Column ( new   DefinedCol  umnSchema (
DEFINED_CO  L_NAME_3 ,  DefinedCol  umnType . INTEGER )); //  You
can   specify   a   pre - defined   attribute   for   the   base
table .
```

```
    int   timeToLive  = - 1 ; //  You   can   specify  - 1   as
the   Time   To   Live  ( TTL )  value   so   the   data   never
expires .
    int   maxVersion  s  =  1 ; //  The   maximum   version
number .  You   can   only   specify   1   as   the   version
value   when   a   base   table   have   one   or   more   indexes
.

    TableOptio  ns   tableOptio  ns  =  new   TableOptio  ns (
timeToLive ,  maxVersion  s );

    ArrayList < IndexMeta >  indexMetas  =  new   ArrayList <
IndexMeta >();
    IndexMeta   indexMeta  =  new   IndexMeta ( INDEX_NAME ); //
You   can   create   an   index .
    indexMeta . addPrimary  KeyColumn ( DEFINED_CO  L_NAME_1 ); //
You   can   specify   DEFINED_CO  L_NAME_1   of   the   base
table   as   an   index   primary   key .
    indexMeta . addDefined  Column ( DEFINED_CO  L_NAME_2 ); //
You   can   specify   DEFINED_CO  L_NAME_2   of   the   base
table   as   an   index   primary   key .
    indexMetas . add ( indexMeta ); //  You   can   add   the
index   to   the   base   table .

    CreateTabl  eRequest   request  =  new   CreateTabl  eRequest (
tableMeta ,  tableOptio  ns ,  indexMetas ); //  You   can   create
  the   base   table .

    client . createTabl  e ( request );
}
```

- You can create an index on a base table.

```
private   static   void   createInde  x ( SyncClient   client ) {
    IndexMeta   indexMeta  =  new   IndexMeta ( INDEX_NAME ); //
Create   index   meta .
    indexMeta . addPrimary  KeyColumn ( DEFINED_CO  L_NAME_2 ); //
Specify   DEFINED_CO  L_NAME_2   as   the   first   primary   key
column   of   the   index   table .
    indexMeta . addPrimary  KeyColumn ( DEFINED_CO  L_NAME_1 ); //
Specify   DEFINED_CO  L_NAME_2   as   the   second   primary   key
  column   of   the   index   table .
    CreateInde  xRequest   request  =  new   CreateInde  xRequest (
TABLE_NAME ,  indexMeta ,  true ); //  Add   the   index   table
to   the   source   table ,  including   stock   data
    CreateInde  xRequest   request  =  new   CreateInde  xRequest (
TABLE_NAME ,  indexMeta ,  false ); //  Add   the   index   table
to   the   source   table ,  not   including   stock   data
    client . createInde  x ( request ); //  Create   an   index
table .
}
```

Note:

At the moment, existing data in the base table will not be copied to the index when you create an index on a base table. The newly created index only includes incremental data after you create this index. For more information about incremental data, contact Table Store technical support with DingTalk.

- You can delete an index.

```
private   static   void   deleteInde  x ( SyncClient   client ) {
    DeleteInde  xRequest   request  =  new   DeleteInde  xRequest (
TABLE_NAME ,  INDEX_NAME ); //  You   can   specify   the   names
of   a   base   table   and   an   index .
    client . deleteInde  x ( request ); //  You   can   delete   an
  index .
}
```

- You can read data from an index.

  If an index includes an attribute that will be returned in results, you can directly retrieve data from the index.

```
private   static   void   scanFromIn  dex ( SyncClient   client ) {
    RangeRowQu  eryCriteri  a   rangeRowQu  eryCriteri  a  =  new
RangeRowQu  eryCriteri  a ( INDEX_NAME ); //  You   can   specify
the   name   of   an   index .

   //  You   can   specify   the   start   primary   key .
    PrimaryKey  Builder   startPrima  ryKeyBuild  er  =
PrimaryKey  Builder . createPrim  aryKeyBuil  der ();
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
DEFINED_CO  L_NAME_1 ,  PrimaryKey  Value . INF_MIN ); //  You
can   specify   the   minimum   value   for   an   index   primary
  key .
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_1 ,  PrimaryKey  Value . INF_MIN ); //  You
can   specify   the   minimum   value   for   a   base   table
primary   key .
    startPrima  ryKeyBuild  er . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME_2 ,  PrimaryKey  Value . INF_MIN ); //  You
can   specify   the   minimum   value   for   a   base   table
primary   key .
    rangeRowQu  eryCriteri  a . setInclusi  veStartPri  maryKey (
startPrima  ryKeyBuild  er . build ());

   //  You   can   specify   the   end   primary   key .
    PrimaryKey  Builder   endPrimary  KeyBuilder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( DEFINED_CO
L_NAME_1 ,  PrimaryKey  Value . INF_MAX ); //  You   can   specify
  the   maximum   value   for   an   index   attribute .
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_1 ,  PrimaryKey  Value . INF_MAX ); //  You   can   specify
  the   maximum   value   for   a   base   table   primary   key .
    endPrimary  KeyBuilder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME_2 ,  PrimaryKey  Value . INF_MAX ); //  You   can   specify
  the   maximum   value   for   a   base   table   primary   key .
    rangeRowQu  eryCriteri  a . setExclusi  veEndPrima  ryKey (
endPrimary  KeyBuilder . build ());

    rangeRowQu  eryCriteri  a . setMaxVers  ions ( 1 );

    System . out . println (" The   results   returned   from   an
  index   are   as   follows :");
    while  ( true ) {
        GetRangeRe  sponse   getRangeRe  sponse  =  client .
getRange ( new   GetRangeRe  quest ( rangeRowQu  eryCriteri  a ));
        for  ( Row   row  :  getRangeRe  sponse . getRows ()) {
```

```
            System . out . println ( row );
        }

        // If the nextStartP rimaryKey value is not
 null , you can continue to read data from the base
  table .
         if ( getRangeRe sponse . getNextSta rtPrimaryK ey () !
 = null ) {
            rangeRowQu eryCriteri a . setInclusi veStartPri
 maryKey ( getRangeRe sponse . getNextSta rtPrimaryK ey ());
        } else {
            break ;
        }
    }
}
```

If an index does not include an attribute that will be returned in results, you must query the base table.

```
 private static void scanFromIn dex ( SyncClient client ) {
     RangeRowQu eryCriteri a rangeRowQu eryCriteri a = new
 RangeRowQu eryCriteri a ( INDEX_NAME ); // You can specify
 the index name .

    // You can specify the start primary key .
     PrimaryKey Builder startPrima ryKeyBuild er =
 PrimaryKey Builder . createPrim aryKeyBuil der ();
     startPrima ryKeyBuild er . addPrimary KeyColumn (
 DEFINED_CO L_NAME_1 , PrimaryKey Value . INF_MIN ); // You
  can specify the minimum value for an indexed
 attribute of an index .
     startPrima ryKeyBuild er . addPrimary KeyColumn (
 PRIMARY_KE Y_NAME_1 , PrimaryKey Value . INF_MIN ); // You
 can specify the minimum value for a primary key
 of a base table .
     startPrima ryKeyBuild er . addPrimary KeyColumn (
 PRIMARY_KE Y_NAME_2 , PrimaryKey Value . INF_MIN ); // You
 can specify the minimum value for a primary key
 of a base table .
     rangeRowQu eryCriteri a . setInclusi veStartPri maryKey (
 startPrima ryKeyBuild er . build ());

    // You can specify the end primary key .
     PrimaryKey Builder endPrimary KeyBuilder = PrimaryKey
 Builder . createPrim aryKeyBuil der ();
     endPrimary KeyBuilder . addPrimary KeyColumn ( DEFINED_CO
 L_NAME_1 , PrimaryKey Value . INF_MAX ); // You can specify
  the maximum value for an indexed attribute of an
  index .
     endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KE
 Y_NAME_1 , PrimaryKey Value . INF_MAX ); // You can specify
  the maximum value for a base table primary key .
     endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KE
 Y_NAME_2 , PrimaryKey Value . INF_MAX ); // You can specify
  the maximum value for a base table primary key .
     rangeRowQu eryCriteri a . setExclusi veEndPrima ryKey (
 endPrimary KeyBuilder . build ());

     rangeRowQu eryCriteri a . setMaxVers ions ( 1 );

     while ( true ) {
```

```
        GetRangeRe sponse getRangeRe sponse = client .
getRange ( new GetRangeRe quest ( rangeRowQu eryCriteri a ));
        for ( Row row : getRangeRe sponse . getRows ()) {
            PrimaryKey curIndexPr imaryKey = row . getPrimary
Key ();
            PrimaryKey Column pk1 = curIndexPr imaryKey .
getPrimary KeyColumn ( PRIMARY_KE Y_NAME1 );
            PrimaryKey Column pk2 = curIndexPr imaryKey .
getPrimary KeyColumn ( PRIMARY_KE Y_NAME2 );
            PrimaryKey Builder mainTableP KBuilder =
PrimaryKey Builder . createPrim aryKeyBuil der ();
            mainTableP KBuilder . addPrimary KeyColumn (
PRIMARY_KE Y_NAME1 , pk1 . getValue ());
            mainTableP KBuilder . addPrimary KeyColumn (
PRIMARY_KE Y_NAME2 , ke2 . getValue ());
            PrimaryKey mainTableP K = mainTableP KBuilder .
build (); // You can specify the index primary keys
for a base table .

           // You can query a base table .
            SingleRowQ ueryCriter ia criteria = new
SingleRowQ ueryCriter ia ( TABLE_NAME , mainTableP K );
            criteria . addColumns ToGet ( DEFINED_CO L_NAME3
); // You can read the DEFINED_CO L_NAME3 attribute
from the base table .
           // You can retrieve the latest data version
.
            criteria . setMaxVers ions ( 1 );
            GetRowResp onse getRowResp onse = client . getRow
( new GetRowRequ est ( criteria ));
            Row mainTableR ow = getRowResp onse . getRow ();
            System . out . println ( row );
       }

       // If the value of nextStartP rimaryKey is not
  null , you can continue to read data from the
base table .
        if ( getRangeRe sponse . getNextSta rtPrimaryK ey () !
= null ) {
            rangeRowQu eryCriteri a . setInclusi veStartPri
maryKey ( getRangeRe sponse . getNextSta rtPrimaryK ey ());
       } else {
            break ;
       }
    }
}
```

7.5 APIs

CreateTable

You can call the `CreateTable` method to create a table, and an index with pre-defined attributes at the same time.

When you write data to a base table, an index on this base table is also updated. For more information, see CreateTable.

CreateIndex

You can call the `CreateIndex` method to create an index on a base table.

Note:
The current version does not support copying existing base table data to the index when you call the CreateIndex method to create an index on a base table. This feature will be supported by later versions.

DeleteIndex

You can call the `DeleteIndex` method to delete indexes on a base table. The other indexes on this table will not be affected.

DeleteTable

You can call the `DeleteTable` method to delete a base table and all indexes on this table. For more information, see DeleteTable.

7.6 Billing rules

To use secondary indexes, index tables are needed. Therefore, additional storage space is required to store index tables. When the system inserts data to a primary table, it may also need to write the index tables created on the primary table at the same time. During this process, read and write CUs are consumed. This topic describes the billing rules for secondary indexes.

Note:
Capacity units (CUs) are read and write throughput units. They are the smallest units used to measure the costs of read and write operations For example, when the system reads 4 KB from one row per second, one read CU is consumed.

To use secondary indexes, index tables are needed. Therefore, additional storage space is required to store index tables. When the system inserts data to a primary table, it may also need to write the index tables created on the primary table at the same time. During this process, read and write CUs are consumed.

Secondary index billing includes the following parts: the number of read and write CUs consumed to write index tables, the amount of data stored in the index tables, and the amount of data that is read from the index tables.

Billing item	Description
Data storage	The storage space used to store a primary table and its index tables.
Read CUs consumed to write index tables	The number of CUs that are consumed by read operations to delete, insert, or update index rows.
Write CUs consumed to write index tables	The number of CUs that are consumed to insert or update index rows.
CUs consumed by regular read operations	The number of CUs that are consumed to read data from a primary table or index tables using an API.
CUs consumed by regular write operations	The number of CUs that are consumed to insert data to a primary table using an API.

Billing rules for storing, writing, and reading an index table:

- The billing rules for storing and reading an index table are the same as those of a primary table. For more information, see Billing items and pricing.
- CUs are consumed based on the following rules when the system writes an index table:
  - Write CUs are consumed only when an index row is inserted or updated.
  - Read CUs are consumed when an index row is deleted, updated, or inserted. The number of read CUs equals the amount of data read from the corresponding indexed columns in the primary table.

Calculate the number of read CUs consumed to write index tables

When you create secondary indexes on the primary table, read CUs are consumed based on the following rules:

- When you use the PUT operation to insert a data row to the primary table:
  - The PUT operation does not insert data to the indexed attribute columns in the primary table, which means that no index row is inserted. In this case, one read CU is consumed.
  - The PUT operation inserts data to the indexed attribute columns in the primary table, which means that new index rows are inserted. In this case, one read CU is consumed.

- When you use the PUT operation to overwrite a row in the primary table:
  - The PUT operation does not update the indexed attribute columns in the primary table. In this case, one read CU is consumed.
  - The PUT operation updates the indexed attribute columns in the primary table. In this case, the read CUs are consumed as follows:

    Divide the total amount of data read from the indexed attribute columns by four, excluding primary key columns. The number of consumed CUs equals the calculated value rounded up to the nearest integer. If the total amount is 0 KB, one CU is consumed.
- When you use the UPDATE operation to insert a data row to the primary table:
  - If the UPDATE operation does not insert data to the indexed columns in the primary table, no read CU is consumed.
  - If the UPDATE operation inserts data to the indexed columns in the primary table, one read CU is consumed.
- When you use the UPDATE operation to update a row in the primary table:
  - If the UPDATE operation does not insert data to the indexed attribute columns in the primary table, no read CU is consumed.
  - If the UPDATE operation inserts data to the indexed attribute columns in the primary table, read CUs are consumed based on the following rules:

    Divide the total amount of data read from the indexed columns by four, excluding the primary key columns. The number of consumed CUs equals the calculated value rounded up to the nearest integer. If the total amount is 0 KB, one CU is consumed.
- When you use the Delete operation to delete a row in the primary table, read CUs are consumed based on the following rules:

  Divide the total amount of data read from the indexed columns by four, excluding the primary key columns. The number of consumed CUs equals the calculated value rounded up to the nearest integer. If the total amount is 0 KB, one CU is consumed.
- If the primary table uses primary key auto increment, inserting data to the primary table does not consume any read CUs. Updating a row in a primary table

that uses primary key auto increment consumes read CUs. CUs are calculated based on the same rules as those of the UPDATE operation.

> **Note:**
> We recommend that you use primary key auto increment to insert data to a primary table to decrease the number of CUs that are consumed by index tables.
>
> For primary tables that do not use primary key auto increment, one read CU is consumed if a read operation is performed on the indexed columns, even if no data is retrieved. For primary tables that use primary key auto increment, no read operation is performed on the indexed columns when you insert data. Therefore, no read CU is consumed.

Calculate the number of write CUs

When you insert data to the primary table and create secondary indexes, write CUs are consumed. Write CUs are consumed based on the following rules:

- If you insert a row to the primary table and no data in the index table is updated, no write CUs are consumed.
- If you insert a row to the primary table and a new index row is inserted to the index table, write CUs are consumed. The number of the write CUs is determined by the size of the inserted index row.
- If you insert a row to the primary table and an index row is deleted from the index table, write CUs are consumed. The number of the write CUs is determined by the size of the deleted index row.
- If you insert a row to the primary table and an index row in the index table is updated, write CUs are consumed. The number of the write CUs is determined by the size of the updated index row.
- If you insert a row to the primary table, an index row is deleted from the index table, and another index row is inserted to the index table, write CUs are consumed. The number of the write CUs is determined by the total size of the deleted and inserted index rows.

The detailed rules are as follows:

- When you use the PUT operation to insert a data row to a primary table:
  - The PUT operation does not insert data to the indexed attribute columns in the primary table, which means that no index row is inserted. In this case, no read CU is consumed.
  - The PUT operation inserts data to the indexed attribute columns in the primary table, which means that new index rows are inserted. The write CUs consumed for each index table are calculated as follows:

    Divide the total amount of data in the inserted index row by four. The number of consumed CUs equals the calculated value rounded up to the nearest integer.
- When you use the PUT operation to overwrite a row in the primary table:
  - The PUT operation only updates the indexed primary key columns in the primary table. In this case, no write CUs are consumed.
  - The PUT operation updates the indexed columns in the primary table. The write CUs are consumed based on the following rules:

    All indexes updated by the PUT operation consume a certain number of write CUs, except sparse indexes.
- When you use the UPDATE operation to insert a data row to the primary table:
  - If the UPDATE operation does not insert data to the indexed columns in the primary table, no write CUs are consumed.
  - If the UPDATE operation inserts data to the indexed columns in the primary table, the write CUs consumed for each index table are calculated as follows:
    - If the UPDATE operation inserts a new index row, write CUs are consumed . Divide the total size of the data in the index row by four. The number of consumed CUs equals the calculated value rounded up to the nearest integer.
    - If no index row is inserted, no write CUs are consumed.

- When you use the UPDATE operation to update a row in the primary table:
  - If the UPDATE operation does not update the indexed attribute columns, no write CUs are consumed.
  - If the UPDATE operation updates the indexed attribute columns, write CUs consumed for each index table are calculated based on the following rules:
    - If the index table already contains an index row created based on the row to be updated, delete CUs are consumed. The number of the delete CUs is determined by the size of the indexed primary keys in the deleted index row.
    - If a new index row is inserted based on the updated row, write CUs are consumed. The number of the write CUs is determined by the size of the indexed primary keys in the inserted index row.
    - If the UPDATE operation only updates the attribute data in the existing index row but no new index row is inserted, update CUs are consumed.

    Divide the total amount of data in the index row by four. The number of consumed CUs equals the calculated value rounded up to the nearest integer.
- When you use the DELETE operation to delete a row in the primary table, write CUs are consumed based on the following rules:

  If an index table already contains an index row created based on the row to be deleted, write CUs are consumed. Divide the total amount of the data in the corresponding indexed columns by four, excluding the primary key columns. The consumed write CUs equal the calculated value rounded up to the nearest integer.
- If you insert data to a primary table that uses primary key auto increment, write CUs are consumed. The write CUs are calculated based on the same rules as those of the PUT operation. If you update a row in a primary table that uses primary key auto increment, write CUs are consumed. The write CUs are calculated based on the same rules as those of the UPDATE operation.

Measure index table size

The size of an index table is measured based on the same rule as that of a primary table. The size of an index table equals the total size of all rows. The total size of the rows equals the total size of primary keys and attribute data. For more information, see Data storage.

Calculate the number of CUs consumed to read an index table

When you use an SDK, the console, or other methods, such as a DLA, to read an index table, read CUs are consumed. The number of read CUs are calculated based on the same rules as those of reading a primary table.

Examples

The following example uses a primary table that has two index tables to describe how CUs are consumed under different conditions.

The primary table Table contains two primary key columns PK0 and PK1, and three predefined columns Col0, Col1, and Col2. Two index tables, Index0 and Index1, are created on the primary table. Index0 contains three primary keys Col0, PK0, and PK1 and one attribute column Col2. Index1 contains four primary keys Col1, Col0, PK0, and PK1, and no attribute columns. Use the UPDATE operation to update PK0 and PK1 .

- If the target row does not exist in the primary table:
  - Updating Col3 does not consume read or write CUs.
  - Updating Col1 consumes the following CUs:
    - One read CU
    - No write CUs
  - Updating Col0 and Col1 consumes the following CUs:
    - One read CU
    - Index0 consumes write CUs. The number of the write CUs is determined by the total amount of data inserted to Col0, PK0, and PK1. Index1 consumes write CUs. The number of the write CUs is determined by the total amount of data inserted to Col0, Col1, PK0, and PK1.
- If the target row already exists in the primary table:
  - Updating Col3 does not consume read or write CUs.
  - Updating Col2 consumes the following CUs:
    - Read CUs are consumed. The number of the read CUs is determined by the amount of data read from Col0. If the UPDATE operation inserts data to Col0, one CU is consumed.
    - For Index0, if the UPDATE operation insets data to Col0, Index0 does not consume write CUs. If the UPDATE operation updates the data in Col0, Index0

consumed write CUs. The number of the write CUs is determined by the total amount of data inserted to Col0, PK0, PK1, and Col2. Index1 does not consume write CUs.

- Updating Col1 consumes the following CUs:
  - Read CUs are consumed. The number of the read CUs is determined by the amount of data read from Col0 and Col1. If the total amount is 0 KB, one CU is consumed.
  - Index0 does not consume write CUs. For Index1, if an index row is inserted, write CUs are consumed. The number of the write CUs is determined by the amount of data read from Col0 and inserted to Col1, PK0, and PK1. For Index1 , if no data in Col0 is updated, no index row is inserted and no write CUs are consumed. If the data in Col0 and Col1 is updated, write CUs are consumed to delete the corresponding index row. The number of write CUs is determined by the total amount of data read from Col0, Col1, PK0, and PK1.

7.7 Appendix

You can create tables and indexes as follows:

```
private   static   final   String   TABLE_NAME  = " CallRecord  Table
";
    private   static   final   String   INDEX0_NAM  E  = "
IndexOnBeC  alledNumbe  r ";
    private   static   final   String   INDEX1_NAM  E  = "
IndexOnBas  eStation1 ";
    private   static   final   String   INDEX2_NAM  E  = "
IndexOnBas  eStation2 ";
    private   static   final   String   PRIMARY_KE  Y_NAME_1  = "
CellNumber ";
    private   static   final   String   PRIMARY_KE  Y_NAME_2  = "
StartTime ";
    private   static   final   String   DEFINED_CO  L_NAME_1  = "
CalledNumb  er ";
    private   static   final   String   DEFINED_CO  L_NAME_2  = "
Duration ";
    private   static   final   String   DEFINED_CO  L_NAME_3  = "
BaseStatio  nNumber ";

    private   static   void   createTabl  e ( SyncClient   client ) {
        TableMeta   tableMeta  =  new   TableMeta ( TABLE_NAME );
        tableMeta . addPrimary  KeyColumn ( new   PrimaryKey  Schema
( PRIMARY_KE  Y_NAME_1 ,  PrimaryKey  Type . INTEGER ));
        tableMeta . addPrimary  KeyColumn ( new   PrimaryKey  Schema
( PRIMARY_KE  Y_NAME_2 ,  PrimaryKey  Type . INTEGER ));
        tableMeta . addDefined  Column ( new   DefinedCol  umnSchema
( DEFINED_CO  L_NAME_1 ,  DefinedCol  umnType . INTEGER ));
        tableMeta . addDefined  Column ( new   DefinedCol  umnSchema
( DEFINED_CO  L_NAME_2 ,  DefinedCol  umnType . INTEGER ));
```

```
        tableMeta . addDefined  Column ( new   DefinedCol  umnSchema
( DEFINED_CO  L_NAME_3 ,  DefinedCol  umnType . INTEGER ));

        int   timeToLive  = - 1 ; //  The   time   before   the
data   expires . You   can   specify  - 1   as   the   Time   To
Live  ( TTL )  value   so   the   data   never   expires .  Unit :
seconds .  You   must   specify  - 1   as   the   TTL   value   when
  a   table   has   one   or   more   indexes .
        int   maxVersion  s  =  1 ; //   The   maximum   number   of
  versions .  You   must   specify   1   as   the   value   when   a
  table   has   one   or   more   indexes .

        TableOptio  ns   tableOptio  ns  =  new   TableOptio  ns (
timeToLive ,  maxVersion  s );

        ArrayList < IndexMeta >  indexMetas  =  new   ArrayList <
IndexMeta >();
        IndexMeta   indexMeta0  =  new   IndexMeta ( INDEX0_NAM  E );
        indexMeta0 . addPrimary  KeyColumn ( DEFINED_CO  L_NAME_1 );
        indexMetas . add ( indexMeta0 );
        IndexMeta   indexMeta1  =  new   IndexMeta ( INDEX1_NAM  E );
        indexMeta1 . addPrimary  KeyColumn ( DEFINED_CO  L_NAME_3 );
        indexMeta1 . addPrimary  KeyColumn ( PRIMARY_KE  Y_NAME_2 );
        indexMetas . add ( indexMeta1 );
        IndexMeta   indexMeta2  =  new   IndexMeta ( INDEX2_NAM  E );
        indexMeta2 . addPrimary  KeyColumn ( DEFINED_CO  L_NAME_3 );
        indexMeta2 . addPrimary  KeyColumn ( PRIMARY_KE  Y_NAME_2 );
        indexMeta2 . addDefined  Column ( DEFINED_CO  L_NAME_2 );
        indexMetas . add ( indexMeta2 );

        CreateTabl  eRequest   request  =  new   CreateTabl  eRequest
( tableMeta ,  tableOptio  ns ,  indexMetas );

        client . createTabl  e ( request );
   }
```

8 Tunnel service

8.1 Overview

Tunnel Service is an integrated service for full and incremental data consumption based on Table Store API. It provides you with real-time consumption tunnels for distributed data, including incremental data, full data, and full and incremental data. By creating tunnels for a table, you can easily consume historical data and new data in the table.

Background

Table Store is applicable to applications such as metadata management, time series data monitoring, and message systems. These applications often use incremental data streams or full and incremental data streams to trigger extra operations, including:

- Data synchronization: synchronizes data to a cache, search engine, or data warehouse.
- Event driving: triggers Function Compute, sends a consumption notification, or calls an API operation.
- Stream data processing: connects to a stream-processing engine or a stream- and batch-processing engine.
- Data migration: backs up data to OSS or migrates data to a Table Store capacity instance.

You can use Tunnel Service to easily build efficient and elastic solutions to consume full data, incremental data, and full and incremental data in the preceding scenarios.

Features

The following table lists the features provided by Tunnel Service.

Feature	Description
Tunnels for full and incremental data consumption	Tunnel Service allows you to consume incremental data, full data, and full and incremental data simultaneously.

Feature	Description
Orderly incrementa l data consumption	Tunnel Service distributes incremental data to one or more logical partitions sequentially based on the write time. Data in different partitions can be consumed simultaneously.
Consumption latency monitoring	Tunnel Service allows you to call the DescribeTunnel operation to view the recovery point objective (RPO) information of the consumed data on each client. Tunnel Service also allows you to monitor data consumption of tunnels in the Table Store console.
Horizontal scaling of data consumptio n capabilities	Tunnel Service supports automatic load balancing among logical partitions to accelerate data consumption.

8.2 Features

Tunnel Service is an integrated service for full and incremental data consumption based on Table Store API. Tunnel Service provides the following features:

Tunnels for full and incremental data consumption

Tunnel Service allows you to consume incremental data, full data, and full and incremental data simultaneously.

Orderly incremental data consumption

Tunnel Service distributes incremental data to one or more logical partitions sequentially based on the write time. Data in different partitions can be consumed simultaneously.

Consumption latency monitoring

Tunnel Service allows you to call the DescribeTunnel operation to view the recovery point objective (RPO) information of the consumed data on each client. Tunnel Service also allows you to monitor data consumption of tunnels in the Table Store console.

Horizontal scaling of data consumption capabilities

Tunnel Service supports automatic load balancing among logical partitions. With this feature, you can add more Tunnel Clients to accelerate data consumption.

8.3 Description of the data consumption framework

Tunnel Service uses comprehensive operations of Table Store to consume full and incremental data. You can easily consume and process history data and incremental data in tables.

A Tunnel client is an automatic data consumption framework of Tunnel Service. The Tunnel client regularly checks heartbeats to detect active channels, update status of the Channel and ChannelConnect classes, initialize, run, and terminate data processing tasks.

The Tunnel client supports the following features for processing full and incremental data: load balancing, fault recovery, checkpoints, and partition information synchronization to ensure the sequence of consuming information. The Tunnel client allows you to focus on the processing logic of each record.

The following sections describe the features of the Tunnel client, including automatic data processing, load balancing, and fault tolerance. For more information, see Github to check source code of the Tunnel client.

Automatic data processing

The Tunnel client regularly checks for heartbeats to detect active channels, update status of the Channel and ChannelConnect classes, initialize, run, and terminate data processing tasks. This section describes the data processing logic. For more information, see source code.

1. Initialize resources of the Tunnel client.
    a. Change the status of the Tunnel client from Ready to Started.
    b. Set the HeartbeatTimeout and ClientTag parameters in TunnelWorkerConfig to run the ConnectTunnel task and connect Tunnel Service to obtain the ClientId of the current Tunnel client.
    c. Initialize the ChannelDialer class to create a ChannelConnect task. Each ChannelConnect class corresponds to a Channel class, and the ChannelConnect task records data consumption checkpoints.
    d. Set the Callback parameter for processing data and the CheckpointInterval parameter for specifying the interval of outputting checkpoints in Tunnel

Service. In this way, you can create a data processor that automatically outputs checkpoints.

e. Initialize the TunnelStateMachine class to automatically update the status of the Channel class.

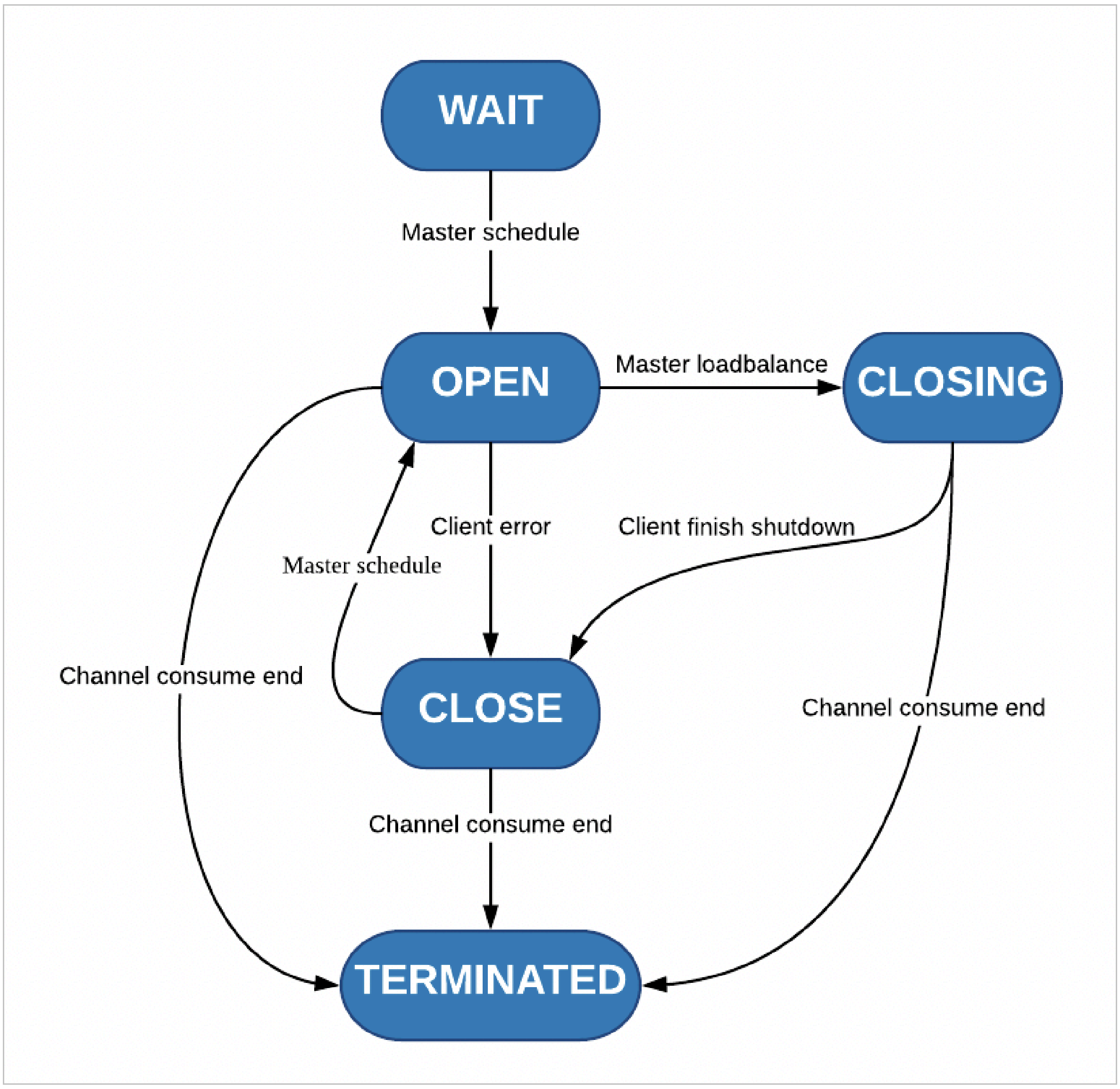
2. Regularly check heartbeat messages.

   You can set the heartbeatIntervalInSec parameter in TunnelWorkerConfig to set the interval for checking the heartbeat.

   a. Send a heartbeat request to obtain the list of latest available channels from Tunnel Service. The list includes the ChannelId, channel versions, and channel status.
   b. Merge the list of channels from Tunnel Service with the local list of channels, and create and update ChannelConnect tasks. Follow these rules:

      - Merge: overwrite the earlier version in the local list with the later version for the same ChannelId from Tunnel Service, and insert the new channels from Tunnel Service into the local list.
      - Create a ChannelConnect task: create a ChannelConnect task in WAIT status for a channel that has no ChannelConnect task. If the ChannelConnect task corresponds to a channel in OPEN status, run the ReadRecords&&ProcessRecords task that cyclically processes data for this ChannelConnect task. For more information, see the ProcessDataPipeline class in source code.
      - Update an existing ChannelConnect task: after you merge the lists of channels, if a channel corresponds to a ChannelConnect task, update the ChannelConnect status according to the status of channels with the same ChannelId. For example, if channels are in Close status, set their ChannelConnect tasks to the Closed status to terminate the corresponding pipeline tasks. For more information, see the ChannelConnect.notifyStatus method in source code.
3. Automatically process channel status.

   Based on the number of active Tunnel clients obtained in the heartbeat request, Tunnel Service allocates available partitions to different clients to balance the loads. Tunnel Service automatically processes channel status as described in the following figure, and drives channel consumption and load balancing.



Tunnel Service and Tunnel clients change their status by using heartbeat requests and channel version updates.

a. Each channel is initially in WAIT status.
b. The channel for incremental data changes to the OPEN status only when the channel consumption on the parent partition is terminated.
c. Tunnel Service allocates the partition in OPEN status to each Tunnel client.
d. During load balancing, Tunnel Service and Tunnel clients use a scheduling protocol for changing a channel status from Open, Closing to Closed. After consuming a BaseData channel or a Stream channel, Tunnel clients report the channel as Terminated.

Automatic load balancing and excellent horizontal scaling

- Multiple Tunnel clients can consume data by using the same Tunnel or TunnelId . When the Tunnel clients run the heartbeat task, Tunnel Service automatically redistributes channels and tries to allocate active channels to each Tunnel client to achieve load balancing.
- You can easily add Tunnel clients to scale out. Tunnel clients can run on one or more instances.

Automatic resource clearing and fault tolerance

- Resource clearing: if Tunnel clients do not shut down normally, such as exceptional exit or manual termination, the system recycles resources automatically. For example, the system can release the thread pool, call the shutdown method that you have registered for the corresponding channel, and terminate the connection to Tunnel Service.
- Fault tolerance: when a Tunnel client has non-parametric errors such as heartbeat timeout, the system automatically renews connections to continue stable data consumption.

8.4 Quick start

You can use Tunnel Service in the Table Store console.

Prerequisites

You have activated Table Store.

Create a tunnel

1. Log on to the Table Store console.
2. Locate the target table and click Tunnels in the Actions column.
3. On the Tunnels page, click Create Tunnel in the upper-right corner.
4. In the Create Tunnel dialog box that appears, set Tunnel Name and Type.

   Tunnel Service provides three types of real-time consumption tunnels for distributed data, including Incremental, Full, and Differential. You can set the type as required. This topic uses the Incremental type as an example.

   After the tunnel is created, you can check the data in the tunnel, monitor consumption latency, and check the number of consumed rows in each channel on the Tunnels page.

Preview data types in a channel

1. In the Table Store console, click Data Editor in the left-side navigation pane. On the Table Data page that appears, click Insert or Delete in the upper-right corner to write or delete data, respectively.
2. Click Tunnels in the left-side navigation pane. On the Tunnels page that appears, locate the tunnel that you created and click Show Channels in the Actions column. The channels are listed at the bottom of the page.
3. Locate the target channel and click View Simulated Export Records in the Actions column. In the dialog box that appears, click Start. The data types in the channel appear.

Enable data consumption for a tunnel

1. Copy a tunnel ID from the tunnel list.
2. Use the Tunnel Service SDK in any programming language to enable data consumption for the tunnel.

```
//  Customize   the   data   consumptio  n   callback ,  that   is
 ,  implement   the   process   and   shutdown   methods   of   the
   IChannelPr  ocessor   interface .
 private   static   class   SimpleProc  essor   implements
 IChannelPr  ocessor  {
    @ Override
     public   void   process ( ProcessRec  ordsInput   input ) {
         System . out . println (" Default   record   processor ,
 would   print   records   count ");
         System . out . println (
             String . format (" Process  % d   records ,  NextToken
 : % s ",  input . getRecords (). size (),  input . getNextTok  en
 ()));
         try  {
            //  Mock   record   processing .
             Thread . sleep ( 1000 );
        }  catch  ( Interrupte  dException   e ) {
             e . printStack  Trace ();
        }
    }
    @ Override
     public   void   shutdown () {
         System . out . println (" Mock   shutdown ");
    }
}

//  TunnelWork  erConfig   contains   more   advanced   parameters
 .  For   more   informatio  n ,  see   the   descriptio  n   in
 the   related   topic .
 TunnelWork  erConfig   config  =  new   TunnelWork  erConfig ( new
   SimpleProc  essor ());
//  Configure   TunnelWork  er   and   start   automatic   data
 processing .
 TunnelWork  er   worker  =  new   TunnelWork  er ($ tunnelId ,
 tunnelClie  nt ,  config );
```

```
 try  {
     worker . connectAnd  Working ();
}  catch  ( Exception   e ) {
     e . printStack  Trace ();
     worker . shutdown ();
     tunnelClie  nt . shutdown ();
}
```

View data consumption logs

You can view the consumption logs of incremental data in the data consumption standard output. You can also log on to the Table Store console or call the DescribeTu nnel operation to view the consumption latency and the number of consumed rows in each channel.

8.5 SDKs

You can use the following SDKs to implement Tunnel Service:

- Go SDK
- Java SDK

8.6 Incremental synchronization performance white paper

This topic describes the test on the performance of incremental synchronization through Tunnel Service, including the test environment, tools, plan, indicators, results, and summary.

Test environment

- Table Store instance
  - Type: high-performance instance
  - Region: China (Hangzhou)
  - Address: a private IP address, which prevents interference caused by unknown network issues.

- Test server configuration
  - Type: Alibaba Cloud ECS
  - Region: China (Hangzhou)
  - Model: ecs.mn4.4xlarge balanced entry-level model
  - Configuration:
    - CPU: 16 cores
    - Memory: 64 GB
    - NIC: VirtIO network device of Red Hat, Inc.
    - Operating system: CentOS 7u2

Test tools

- Stress testing tool

  The stress testing tool of Table Store is used to write data to multiple rows simultaneously by calling the BatchWriteRow operation through the Table Store Java SDK.
- Pre-splitting tool

  The stress testing tool of Table Store is used to automatically create and pre-split tables based on the configured table names and the number of partitions.
- Rate statistics tool

  The Table Store Java SDK can collect statistics of the consumption rate of incremental data and the total number of consumed rows in real time. You can add the logic demonstrated in the following example to the callback to collect rate statistics.

  Example

```
private   static   final   Gson   GSON  =  new   Gson ();
    private   static   final   int   CAL_INTERV  AL_MILLIS  =  5000
;
    static   class   PerfProces  sor   implements   IChannelPr
ocessor  {
        private   static   final   AtomicLong   counter  =  new
AtomicLong ( 0 );
        private   static   final   AtomicLong   latestTs  =  new
AtomicLong ( 0 );
        private   static   final   AtomicLong   allCount  =  new
AtomicLong ( 0 );

       @ Override
        public   void   process ( ProcessRec  ordsInput   input ) {
            counter . addAndGet ( input . getRecords (). size ());
            allCount . addAndGet ( input . getRecords (). size ());
```

```
            if  ( System . currentTim  eMillis () -  latestTs . get
() >  CAL_INTERV  AL_MILLIS ) {
                synchroniz  ed  ( PerfProces  sor . class ) {
                    if  ( System . currentTim  eMillis () -
latestTs . get () >  CAL_INTERV  AL_MILLIS ) {
                        long   seconds  =  TimeUnit . MILLISECON
DS . toSeconds ( System . currentTim  eMillis () -  latestTs . get
());
                        PerfElemen  t   element  =  new
PerfElemen  t ( System . currentTim  eMillis (),  counter . get
() /  seconds ,  allCount . get ());
                        System . out . println ( GSON . toJson (
element ));
                        counter . set ( 0 );
                        latestTs . set ( System . currentTim
eMillis ());
                    }
                }
            }
        }

        @ Override
         public   void   shutdown () {
            System . out . println (" Mock   shutdown ");
        }
    }
```

Test plan

When Tunnel Service is used for data synchronization, it synchronizes data sequentially within a single channel to maintain the order of data, and synchronizes data in different channels in parallel. For incremental data, the number of channels is equal to the number of partitions in a table. This performance test focuses on how the number of partitions (channels) affects the incremental synchronization rate because the overall performance of Tunnel Service is greatly correlated with the number of partitions.

- Test scenarios

  The test is conducted in the following scenarios:

  - Single-server single-partition synchronization
  - Single-server 4-partition synchronization
  - Single-server 8-partition synchronization
  - Single-server 32-partition synchronization
  - Single-server 64-partition synchronization
  - Double-server 64-partition synchronization
  - Double-server 128-partition synchronization

  Note:
  The test in the preceding scenarios is not an extreme test of the service performance, and therefore does not impose much pressure on the Table Store instance.

- Test procedure

  1. Create and pre-split a table for each test scenario.
  2. Create a tunnel for incremental synchronization.
  3. Use the stress testing tool to write incremental data.
  4. Use the rate statistics tool to measure the QPS in real time, and check the consumption of system resources, such as CPU and memory.
  5. Check the total bandwidth consumed during the incremental synchronization.

- Test data description

  ample data includes four primary key columns and one or two attribute columns . The size of each row is approximately 220 bytes. The first primary key (partition key) is a 4-byte hash value, which eguarantees that stress testing data is evenly written to each partition.

Test indicators

This test uses the following indicators:

- QPS (row): the number of rows synchronized per second.
- Average latency (ms per 1,000 rows): the time required to synchronize 1,000 rows, in milliseconds.
- CPU (core): the total number of single-core CPUs used for data synchronization.

- Memory (GB): the total physical memory used for data synchronization.
- Bandwidth (Mbit/s): the total bandwidth used for data synchronization.

Note:
This performance test is based on user experience, rather than extreme testing.

Test results

This section describes the test results for each scenario. For more information, see test details.

- QPS and latency

  The following figure shows the number of rows synchronized per second and the time required to synchronize 1,000 rows in each scenario. In this figure, the QPS increases linearly with the number of partitions.

  In the single-server 64-partition synchronization scenario, the gigabit NIC works at its full capacity, resulting in only 570,000 QPS. For more information, see test details. The QPS in the double-server 64-partition synchronization scenario reaches 780,000, which is approximately twice the 420,000 QPS in the single-server 32-partition synchronization scenario. In the double-server 128-partition synchronization scenario, the QPS reaches 1,000,000.
- System resource consumption

  The following figure shows the CPU and memory usage in each scenario. The CPU usage increases linearly with the number of partitions.

  The single-server single-partition synchronization uses 0.25 single-core CPUs. When the QPS reaches 1,000,000 in the double-server 128-partition synchronization scenario, only 10.2 single-core CPUs are used. The memory usage increases linearly with the number of partitions when it is less than 32. When more partitions, for example, 32, 64, or 128 partitions in this test, need to be processed, the memory usage is stably around 5.3 GB on each server.

- Total bandwidth consumption

  The following figure shows the total bandwidth consumed during the incrementa l synchronization. In this figure, the consumed bandwidth increases linearly with the number of partitions.

  The single-server 64-partition synchronization uses a total bandwidth of 125 Mbit /s, which is the maximum rate supported by the gigabit NIC. In the double-server 64-partition synchronization scenario, the consumed bandwidth is 169 Mbit/s, which is the actual bandwidth required for 64-partition synchronization. This is approximately twice the 86 Mbit/s bandwidth required in the single-server 32-partition synchronization scenario. When the QPS reaches 1,000,000 in the double -server 128-partition synchronization scenario, the total bandwidth consumed reaches 220 Mbit/s.

Test details

- Single-server single-channel: 19,000 QPS.
  - Tested at: 17:40, January 30, 2019.
  - QPS: steady at approximately 19,000 rows per second, with a peak rate of 21,800 rows per second.
  - Latency: approximately 50 ms per 1,000 rows.
  - CPU usage: approximately 25% of a single-core CPU.
  - Memory usage: approximately 0.4% of the total physical memory, which is approximately 0.256 GB. (Each test server provides 64 GB physical memory.)
  - Bandwidth consumption: approximately 4,000 Kbit/s.
- Single-server 4-partition synchronization: 70,000 QPS.
  - Tested at: 20:00, January 30, 2019.
  - QPS: steady at approximately 70,000 rows per second, with a peak rate of 72,400 rows per second.
  - Latency: approximately 14.28 ms per 1,000 rows.
  - CPU usage: approximately 70% of a single-core CPU.
  - Memory usage: approximately 1.9% of the total physical memory, which is approximately 1.1 GB. (Each test server provides 64 GB physical memory.)
  - Bandwidth consumption: approximately 13 Mbit/s.

- Single-server 8-partition synchronization: 130,000 QPS.
  - Tested at: 20:20, January 30, 2019.
  - QPS: steady at approximately 130,000 rows per second, with a peak rate of 141,644 rows per second.
  - Latency: approximately 7.69 ms per 1,000 rows.
  - CPU usage: approximately 120% of a single-core CPU.
  - Memory usage: approximately 4.1% of the total physical memory, which is approximately 2.62 GB. (Each test server provides 64 GB physical memory.)
  - Bandwidth consumption: approximately 27 Mbit/s.
- Single-server 32-partition synchronization: 420,000 QPS.
  - Tested at: 15:50, January 31, 2019.
  - QPS: steady at approximately 420,000 rows per second, with a peak rate of 447,600 rows per second.
  - Latency: 2.38 ms per 1,000 rows.
  - CPU usage: approximately 450% of a single-core CPU.
  - Memory usage: approximately 8.2% of the total physical memory, which is approximately 5.25 GB. (Each test server provides 64 GB physical memory.)
  - Bandwidth consumption: approximately 86 Mbit/s.
- Single-server 64-partition synchronization: 570,000 QPS, with the gigabit NIC working at its full capacity.
  - Tested at: 22:10, January 31, 2019.
  - QPS: steady at approximately 570,000 rows per second, with a peak rate of 581,400 rows per second.
  - Latency: approximately 1.75 ms per 1,000 rows.
  - CPU usage: approximately 640% of a single-core CPU.
  - Memory usage: approximately 8.4% of the total physical memory, which is approximately 5.376 GB. (Each test server provides 64 GB physical memory.)
  - Bandwidth consumption: approximately 125 Mbit/s, which is the maximum rate of the gigabit NIC.

- Double-server 64-partition synchronization: 780,000 QPS.
  - Tested at: 22:30, January 31, 2019.
  - QPS: steady at approximately 390,000 rows per second on each server and 780,000 rows per second on both servers.
  - Latency: approximately 1.28 ms per 1,000 rows.
  - CPU usage: approximately 420% of a single-core CPU on each server and 840% of a single-core CPU on both servers.
  - Memory usage: approximately 8.2% of the total physical memory, which is approximately 10.5 GB. (Each test server provides 64 GB physical memory.)
  - Bandwidth consumption: approximately 169 Mbit/s. This indicates that bandwidth becomes the bottleneck when the number of partitions reaches 64 in single-server scenarios.
- Double-server 128-partition synchronization: 1,000,000 QPS, with both gigabit NICs almost working at their full capacities.
  - Tested at: 23:20, January 31, 2019.
  - QPS: steady at approximately 500,000 rows per second on each server and 1,000,000 rows per second on both servers.
  - Latency: approximately 1 ms per 1,000 rows.
  - CPU usage: approximately 560% of a single-core CPU on each server and 1,020% of a single-core CPU on both servers.
  - Memory usage: approximately 8.2% of the total physical memory, which is approximately 10.5 GB. (Each test server provides 64 GB physical memory.)
  - Bandwidth consumption: approximately 220 Mbit/s.

Summary

Based on this performance test for incremental synchronization, the QPS for tables with a single or a few partitions is mainly affected by the latency in data reading and only few resources on the server are consumed. As the number of partitions increases, the overall throughput of incremental synchronization through Tunnel Service increases linearly until the system bottleneck, such as the bandwidth in this test, is encountered. When a resource on a single server is used up, this resource becomes the bottleneck. You can add more servers to increase the overall throughput. This test validates the excellent horizontal scaling performance of Tunnel Service.

9 HBase

9.1 Table Store HBase Client

In addition to SDKs and RESTful APIs, Table Store HBase Client can be used to access Table Store through Java applications built on open source HBase APIs. Based on Java SDKs for Table Store version 4.2.x and later, Table Store HBase Client supports open source APIs for HBase version 1.x.x and later.

Table Store HBase Client can be obtained from any of the following three channels:

- GitHub tablestore-hbase-client project
- Compressed package
- Maven

```
< dependenci  es >
       < dependency >
           < groupId > com . aliyun . openservic  es </ groupId >
           < artifactId > tablestore - hbase - client </ artifactId
>
           < version > 1 . 2 . 0 </ version >
       </ dependency >
   </ dependenci  es >
```

Table Store is a fully managed NoSQL database service. When using TableStore HBase Client, you can simply ignore HBase Server. Instead, you only need to perform table or data operations using APIs exposed by Client.

Compared with self-built HBase services, Table Store has the following advantages:

Items	Table Store	Self-built HBase cluster
Cost	Billing is based on actual data volumes. By providing high performance and capacity instances, Table Store can be tailored to all scenarios.	Allocates resources based on traffic peaks. Resources remain idle during off-peak periods, resulting in high operation and maintenance costs.

Items	Table Store	Self-built HBase cluster
Security	Integrates Alibaba Cloud RAM and supports multiple authentication and authorization mechanisms, VPC, and primary/RAM user account management. Authorization granularity can be defined at both the table-level and API-level.	Requires extra security mechanisms.
Reliability	Supports automatic redundant data backup and failover. Data availability is 99.9% or greater, and data reliability is 99.99999999%.	Is dependent on cluster reliability.
Scalability	Server Load Balancer of Table Store supports PB-level data transfer from a single table. Manual resizing is not needed even if millions of bytes of data is concurrently stored.	Complex online/offline processes are required if a cluster reaches high usage capacity, which can severely impact online services.

9.2 Table Store HBase Client supported functions

API support differences between Table Store and HBase

Table Store and HBase, while similar in terms of data model and functions, have different APIs. The following sections detail differences between Table Store HBase Client APIs and HBase APIs.

Functions supported by Table Store HBase Client APIs:

- CreateTable

  Table Store does not support ColumnFamily as all data can be considered to be in the same ColumnFamily. This means that TTL and Max Versions of Table Store are at the table-level. Therefore, Table Store has some support for the following functions:

Functions	Supported or Not
family max version	Table-level Max Versions supported. Default value: 1
family min version	Unsupported
family ttl	Table-level TTL supported
is/set ReadOnly	Supported through the sub-account of RAM
Pre-partitioning	Unsupported
blockcache	Unsupported
blocksize	Unsupported
BloomFilter	Unsupported
column max version	Unsupported
cell ttl	Unsupported
Control parameter	Unsupported

- Put

Functions	Supported or Not
Writes multiple columns of data at a time	Supported
Specifies a timestamp	Supported
Uses the system time by default if no timestamp is specified	Supported
Single-row ACL	Unsupported
ttl	Unsupported
Cell Visibility	Unsupported
tag	Unsupported

- Get

  Table Store guarantees high data consistency. If the HTTP 200 status code (OK) is returned after data is written to an API, the data is permanently written to all copies, and can be read immediately by Get.

Functions	Supported or Not
Reads a row of data	Supported
Reads all columns in a ColumnFamily	Supported
Reads data from a specified column	Supported
Reads data with a specified timestamp	Supported
Reads data of a specified number of versions	Supported
TimeRange	Supported
ColumnfamilyTimeRange	Unsupported
RowOffsetPerColumnFamily	Supported
MaxResultsPerColumnFamily	Unsupported
checkExistenceOnly	Unsupported
closestRowBefore	Supported
attribute	Unsupported
cacheblock:true	Supported
cacheblock:false	Unsupported
IsolationLevel:READ_COMMITTED	Supported
IsolationLevel:READ_UNCOMMITTED	Unsupported
IsolationLevel:STRONG	Supported
IsolationLevel:TIMELINE	Unsupported

- Scan

  Table Store guarantees high data consistency. If the HTTP 200 status code (OK) is returned after data is written to an API, the data is permanently written to all copies, which can be read immediately by Scan.

Functions	Supported or Not
Determines a scanning range based on the specified start and stop	Supported

Functions	Supported or Not
Globally scans data if no scanning range is specified	Supported
prefix filter	Supported
Reads data using the same logic as Get	Supported
Reads data in reverse order	Supported
caching	Supported
batch	Unsupported
maxResultSize, indicating the maximum size of the returned data volume	Unsupported
small	Unsupported
batch	Unsupported
cacheblock:true	Supported
cacheblock:false	Unsupported
IsolationLevel:READ_COMMITTED	Supported
IsolationLevel:READ_UNCOMMITTED	Unsupported
IsolationLevel:STRONG	Supported
IsolationLevel:TIMELINE	Unsupported
allowPartialResults	Unsupported

- Batch

Functions	Supported or Not
Get	Supported
Put	Supported
Delete	Supported
batchCallback	Unsupported

- Delete

Functions	Supported or Not
Deletes a row	Supported
Deletes all versions of the specified column	Supported

Functions	Supported or Not
Deletes the specified version of the specified column	Supported
Deletes the specified ColumnFamily	Unsupported
When a timestamp is specified, deleteColumn deletes the versions that are equal to the timestamp	Supported
When a timestamp is specified, deleteFamily and deleteColumn delete the versions that are earlier than or equal to the timestamp	Unsupported
When no timestamp is specified, deleteColumn deletes the latest version	Unsupported
When no timestamp is specified, deleteFamily and deleteColumn delete the version of the current system time	Unsupported
addDeleteMarker	Unsupported

- checkAndXXX

Functions	Supported or Not
CheckAndPut	Supported
checkAndMutate	Supported
CheckAndDelete	Supported
Checks whether the value of a column meets the conditions. If yes, checkAndXXX deletes the column.	Supported
Uses the default value if no value is specified	Supported
Checks row A and executes row B.	Unsupported

- Exist

Functions	Supported or Not
Checks whether one or more rows exist and does not return any content	Supported

- Filter

Functions	Supported or Not
ColumnPaginationFilter	columnOffset and count unsupported
SingleColumnValueFilter	Supported: LongComparator, BinaryComparator, and ByteArrayComparable Unsupported: RegexStringComparator, SubstringComparator, and BitComparator

Functions not supported by Table Store HBase Client APIs

- Namespaces

  Table Store uses instances to manage a data table. An instance is the minimum billing unit in Table Store. You can manage instances in the Table Store console. Therefore, the following features are not supported:

  - createNamespace(NamespaceDescriptor descriptor)
  - deleteNamespace(String name)
  - getNamespaceDescriptor(String name)
  - listNamespaceDescriptors()
  - listTableDescriptorsByNamespace(String name)
  - listTableNamesByNamespace(String name)
  - modifyNamespace(NamespaceDescriptor descriptor)
- Region management

  Data partition is the basic unit for data storage and management in Table Store. Table Store automatically splits or merges the data partitions based on their data volumes and access conditions. Therefore, Table Store does not support features related to Region management in HBase.
- Snapshots

  Table Store does not support Snapshots, or related featurs of Snapshots.

- Table management

  Table Store automatically splits, merges, and compacts data partitions in tables. Therefore, the following features are not supported:

  - getTableDescriptor(TableName tableName)
  - compact(TableName tableName)
  - compact(TableName tableName, byte[] columnFamily)
  - flush(TableName tableName)
  - getCompactionState(TableName tableName)
  - majorCompact(TableName tableName)
  - majorCompact(TableName tableName, byte[] columnFamily)
  - modifyTable(TableName tableName, HTableDescriptor htd)
  - split(TableName tableName)
  - split(TableName tableName, byte[] splitPoint)
- Coprocessors

  Table Store does not support the coprocessor. Therefore, the following features are not supported:

  - coprocessorService()
  - coprocessorService(ServerName serverName)
  - getMasterCoprocessors()
- Distributed procedures

  Table Store does not support Distributed procedures. Therefore, the following features are not supported:

  - execProcedure(String signature, String instance, Map props)
  - execProcedureWithRet(String signature, String instance, Map props)
  - isProcedureFinished(String signature, String instance, Map props)
- Increment and Append

  Table Store does not support atomic increase/decrease or atomic Append.

9.3 Differences between Table Store and HBase

This topic introduces features of Table Store HBase Client and explains restricted and supported functions when compared with HBase. Features are listed as follows.

Table

Table Store only supports single ColumnFamilies, that is, it does not support multi-ColumnFamilies.

Row and Cell

- Table Store does not support ACL settings.
- Table Store does not support Cell Visibility settings.
- Table Store does not support Tag settings.

GET

Table Store only supports single ColumnFamilies. Therefore, it does not support ColumnFamily related APIs, including:

- setColumnFamilyTimeRange(byte[] cf, long minStamp, long maxStamp)
- setMaxResultsPerColumnFamily(int limit)
- setRowOffsetPerColumnFamily(int offset)

SCAN

Similar to GET, Table Store does not support ColumnFamily related APIs and cannot be used to set partial optimization APIs, including:

- setBatch(int batch)
- setMaxResultSize(long maxResultSize)
- setAllowPartialResults(boolean allowPartialResults)
- setLoadColumnFamiliesOnDemand(boolean value)
- setSmall(boolean small)

Batch

Table Store does not support BatchCallback.

Mutations and Deletions

- Table Store does not support deletion of the specified ColumnFamily.
- Table Store does not support deletion of the versions with the latest timestamp.
- Table Store does not support deletion of all versions earlier than the specified timestamp.

Increment and Append

Table Store does not support Increment or Append features.

Filter

- Table Store supports ColumnPaginationFilter.
- Table Store supports FilterList.
- Table Store partially supports SingleColumnValueFilter, and supports only BinaryComparator.
- Table Store does not support other Filters.

Optimization

Some of the HBase APIs involve access and storage optimization. These APIs are not opened currently:

- blockcache: The default value is "true", which cannot be modified.
- blocksize: The default value is "64 KB", which cannot be modified.
- IsolationLevel: The default value is "READ_COMMITTED", which cannot be modified.
- Consistency: The default value is "STRONG", which cannot be modified.

Admin

The `org . apache . hadoop . hbase . client . Admin` APIs of HBase are used for management and control, most of which are not required in Table Store.

As Table Store is a cloud service, it automatically performs operations such as operation and maintenance, management, and control, which does not need to be concerned. Table Store currently does not support a few of APIs.

- CreateTable

  Table Store only supports single ColumnFamilies. Therefore, you can create only one ColumnFamily when creating a table. The ColumnFamily supports the MaxVersions and TimeToLive parameters.

- Maintenance task

  In Table Store, the following APIs related to task maintenance are automatically processed:

  - abort(String why, Throwable e)
  - balancer()
  - enableCatalogJanitor(boolean enable)
  - getMasterInfoPort()
  - isCatalogJanitorEnabled()
  - rollWALWriter(ServerName serverName) -runCatalogScan()
  - setBalancerRunning(boolean on, boolean synchronous)
  - updateConfiguration(ServerName serverName)
  - updateConfiguration()
  - stopMaster()
  - shutdown()
- Namespaces

  In Table Store, the instance name is similar to Namespaces in HBase. Therefore, it does not support Namespaces related APIs, including:

  - createNamespace(NamespaceDescriptor descriptor)
  - modifyNamespace(NamespaceDescriptor descriptor)
  - getNamespaceDescriptor(String name)
  - listNamespaceDescriptors()
  - listTableDescriptorsByNamespace(String name)
  - listTableNamesByNamespace(String name)
  - deleteNamespace(String name)

- Region

  Table Store automatically performs Region related operations. Therefore, it does not support the following APIs:

  - assign(byte[] regionName)
  - closeRegion(byte[] regionname, String serverName)
  - closeRegion(ServerName sn, HRegionInfo hri)
  - closeRegion(String regionname, String serverName)
  - closeRegionWithEncodedRegionName(String encodedRegionName, String serverName)
  - compactRegion(byte[] regionName)
  - compactRegion(byte[] regionName, byte[] columnFamily)
  - compactRegionServer(ServerName sn, boolean major)
  - flushRegion(byte[] regionName)
  - getAlterStatus(byte[] tableName)
  - getAlterStatus(TableName tableName)
  - getCompactionStateForRegion(byte[] regionName)
  - getOnlineRegions(ServerName sn)
  - majorCompactRegion(byte[] regionName)
  - majorCompactRegion(byte[] regionName, byte[] columnFamily)
  - mergeRegions(byte[] encodedNameOfRegionA, byte[] encodedNameOfRegionB, boolean forcible)
  - move(byte[] encodedRegionName, byte[] destServerName)
  - offline(byte[] regionName)
  - splitRegion(byte[] regionName)
  - splitRegion(byte[] regionName, byte[] splitPoint)
  - stopRegionServer(String hostnamePort)
  - unassign(byte[] regionName, boolean force)

Snapshots

Table Store does not support Snapshots related APIs.

Replication

Table Store does not support Replication related APIs.

Coprocessors

Table Store does not support Coprocessors related APIs.

Distributed procedures

Table Store does not support Distributed procedures related APIs.

Table Management

Table Store automatically performs Table related operations, which does not need to be concerned. Therefore, Table Store does not support the following APIs:

- compact(TableName tableName)
- compact(TableName tableName, byte[] columnFamily)
- flush(TableName tableName)
- getCompactionState(TableName tableName)
- majorCompact(TableName tableName)
- majorCompact(TableName tableName, byte[] columnFamily)
- modifyTable(TableName tableName, HTableDescriptor htd)
- split(TableName tableName)
- split(TableName tableName, byte[] splitPoint)

Restrictions

As Table Store is a cloud service, to guarantee the optimal overall performance, some parameters are restricted and cannot be reconfigured. For more information about the restrictions, see #unique_134.

9.4 Migrate from HBase to Table Store

The following information explains how to migrate HBase to Table Store.

Dependencies

Table Store HBase Client v1.2.0 depends on HBase Client v1.2.0 and Table Store Java SDK v4.2.1. The configuration of `pom . xml` is as follows.

```
< dependenci  es >
       < dependency >
           < groupId > com . aliyun . openservic  es </ groupId >
           < artifactId > tablestore - hbase - client </ artifactId >
           < version > 1 . 2 . 0 </ version >
       </ dependency >
```

```
    </ dependenci  es >
```

If you want to use another HBase Client or Table Store Java SDK version, you must use the exclusion tag. In the following example, HBase Client v1.2.1 and Table Store Java SDK v4.2.0 are used.

```
    < dependenci  es >
        < dependency >
            < groupId > com . aliyun . openservic  es </ groupId >
            < artifactId > tablestore - hbase - client </ artifactId >
            < version > 1 . 2 . 0 </ version >
            < exclusions >
                < exclusion >
                    < groupId > com . aliyun . openservic  es </
  groupId >
                    < artifactId > tablestore </ artifactId >
                </ exclusion >
                < exclusion >
                    < groupId > org . apache . hbase </ groupId >
                    < artifactId > hbase - client </ artifactId >
                </ exclusion >
            </ exclusions >
        </ dependency >
        < dependency >
            < groupId > org . apache . hbase </ groupId >
            < artifactId > hbase - client </ artifactId >
            < version > 1 . 2 . 1 </ version >
        </ dependency >
        < dependency >
            < groupId > com . aliyun . openservic  es </ groupId >
            < artifactId > tablestore </ artifactId >
            < classifier > jar - with - dependenci  es </ classifier >
            < version > 4 . 2 . 0 </ version >
        </ dependency >
    </ dependenci  es >
```

Table Store HBase Client v1.2.x is only compatible with HBase Client v1.2.x, because API changes exist in HBase Client v1.2.x and earlier.

If you want to use HBase Client version v1.1.x, use Table Store HBase Client version v1.1.x.

If you want to use HBase Client version v0.x.x, see Migrate HBase of an earlier version.

Configure the file

To migrate data from HBase Client to Table Store HBase Client, modify the following two items in the configuration file.

- HBase Connection type

  Set Connection to TableStoreConnection.

```
    < property >
```

```
        < name > hbase . client . connection . impl </ name >
        < value > com . alicloud . tablestore . hbase . Tablestore
Connection </ value >
    </ property >
```

- Configuration items of Table Store

  Table Store is a cloud service and provides strict permission management. Table Store offers strict permission management. To access Table Store, you must configure access information such as the AccessKey.

  - You need to configure the following four items before accessing Table Store:

```
< property >
  < name > tablestore . client . endpoint </ name >
  < value ></ value >
 </ property >
 < property >
  < name > tablestore . client . instancena  me </ name >
  < value ></ value >
 </ property >
 < property >
  < name > tablestore . client . accesskeyi  d </ name >
  < value ></ value >
 </ property >
 < property >
  < name > tablestore . client . accesskeys  ecret </ name >
  < value ></ value >
 </ property >
```

  - Optional items you can configure are as follows.

```
 < property >
  < name > hbase . client . tablestore . family </ name >
  < value > f1 </ value >
 </ property >
 < property >
  < name > hbase . client . tablestore . family .$ tablename </
name >
  < value > f2 </ value >
 </ property >
 < property >
  < name > tablestore . client . max . connection  s </ name >
  < value > 300 </ value >
 </ property >
 < property >
  < name > tablestore . client . socket . timeout </ name >
  < value > 15000 </ value >
 </ property >
 < property >
  < name > tablestore . client . connection . timeout </ name >
  < value > 15000 </ value >
 </ property >
 < property >
  < name > tablestore . client . operation . timeout </ name >
  < value > 2147483647 </ value >
 </ property >
 < property >
  < name > tablestore . client . retries </ name >
  < value > 3 </ value >
```

```
</ property >
```

- hbase.client.tablestore.family and hbase.client.tablestore.family.$tablename
  - Table Store only supports single ColumnFamilies. When you use HBase APIs, you must enter the content of the family.

    `hbase . client . tablestore . family` indicates global configuration, while `hbase . client . tablestore . family .$ tablename` indicates configuration of a single table.
  - Rule: For tables whose names are T, search for `hbase . client . tablestore . family . T` first. If the family does not exist, search for `hbase . client . tablestore . family`. If the family does not exist, use the default value f.
- tablestore.client.max.connections

  Maximum connections. The default value is 300.
- tablestore.client.socket.timeout

  Socket time-out time. The default value is 15 seconds.
- tablestore.client.connection.timeout

  Connection time-out time. The default value is 15 seconds.
- tablestore.client.operation.timeout

  API time-out time. The default value is Integer.MAX_VALUE, indicating that the API never times out.
- tablestore.client.retries

  Number of retries when a request fails. The default value is 3.

9.5 Migrate HBase of an earlier version

Table Store HBase Client supports APIs of HBase Client 1.0.0 and later versions.

Compared with earlier versions, HBase Client 1.0.0 has big changes which are incompatible with HBase Client of earlier versions.

If you use an HBase Client from version 0.x.x (that is, an earlier version than 1.0.0), this topic explains how to integrate your HBase Client version with Table Store.

Connection APIs

HBase 1.0.0 and later versions cancel the HConnection APIs, and instead use the `org . apache . hadoop . hbase . client . Connection Factory` series to provide the Connection APIs and replace ConnectionManager and HConnectionManager with ConnectionFactory.

Creating a Connection API has relatively high cost, however, Connection APIs guarantee thread safety. When using a Connection API, you can generate only one Connection object in the program. Multiple threads can then share this object.

You also need to manage the Connection lifecycle, and close it after use.

The latest code is as follows:

```
 Connection   connection  =  Connection  Factory . createConn  ection
 ( config );
// ...
 connection . close ();
```

TableName series

In HBase version 1.0.0 and earlier, you can use a String-type name when creating a table. For later HBase versions, you can use the `org . apache . hadoop . hbase . TableName`.

The latest code is as follows:

```
 String   tableName  = " MyTable ";
//  or   byte []  tableName  =  Bytes . toBytes (" MyTable ");
 TableName   tableNameO  bj  =  TableName . valueOf ( tableName );
```

Table, BufferedMutator, and RegionLocator APIs

From HBase Client v1.0.0, the HTable APIs are replaced with the Table, BufferedMu tator, and RegionLocator APIs.

- `org . apache . hadoop . hbase . client . Table`: Used to operate reading, writing, and other requests of a single table.
- `org . apache . hadoop . hbase . client . BufferedMu tator`: Used for asynchronous batch writing. This API corresponds to `setAutoFlu sh ( boolean )` of the HTableInterface API of the earlier versions.
- `org . apache . hadoop . hbase . client . RegionLoca tor`: Indicates the table partition information.

The Table, BufferedMutator, and RegionLocator APIs do not guarantee thread safety. However, they are lightweight and can be used to create an object for each thread.

Admin APIs

From HBase Client v1.0.0, HBaseAdmin APIs are replaced by `org . apache . hadoop . hbase . client . Admin`. As Table Store is a cloud service, and most operation and maintenance APIs are automatically processed, most Admin APIs are not supported. For more information, see Differences between Table Store and HBase.

Use the Connection instance to create an Admin instance:

```
Admin  admin  =  connection . getAdmin ();
```

9.6 Hello World

This topic describes how to use Table Store HBase Client to implement a simple Hello World program, and includes the following operations:

- Configure project dependencies.
- Connect Table Store
- Create a table
- Write Data
- Read Data
- Scan data
- Delete a table

Code position

This sample program uses HBase APIs to access Table Store. The complete sample program is located in the Github aliyun-tablestore-hbase-client project. The directory is src/test/java/samples/HelloWorld.java.

Use HBase APIs

- Configure project dependencies

  Configure Maven dependencies as follows.

```
    < dependenci  es >
         < dependency >
             < groupId > com . aliyun . openservic  es </ groupId >
             < artifactId > tablestore - hbase - client </ artifactId
 >
```

```
            < version > 1 . 2 . 0 </ version >
        </ dependency >
    </ dependenci  es >
```

For more information about advanced configurations, see Migrate from HBase to Table Store.

- Configure the file

  Add the following configuration items to hbase-site.xml.

```
< configurat  ion >
    < property >
        < name > hbase . client . connection . impl </ name >
        < value > com . alicloud . tablestore . hbase . Tablestore
 Connection </ value >
    </ property >
    < property >
        < name > tablestore . client . endpoint </ name >
        < value > endpoint </ value >
    </ property >
    < property >
        < name > tablestore . client . instancena  me </ name >
        < value > instance_n  ame </ value >
    </ property >
    < property >
        < name > tablestore . client . accesskeyi  d </ name >
        < value > access_key  _id </ value >
    </ property >
    < property >
        < name > tablestore . client . accesskeys  ecret </ name >
        < value > access_key  _secret </ value >
    </ property >
    < property >
        < name > hbase . client . tablestore . family </ name >
        < value > f1 </ value >
    </ property >
    < property >
        < name > hbase . client . tablestore . table </ name >
        < value > ots_adapto  r </ value >
    </ property >
</ configurat  ion >
```

  For more information about advanced configurations, see Migrate from HBase to Table Store.

- Connect Table Store

  Create a TableStoreConnection object to connect Table Store.

```
   Configurat  ion   config  =  HBaseConfi  guration . create ();

  //  Create   a   Tablestore   Connection
   Connection   connection  =  Connection  Factory . createConn
 ection ( config );

  //  Admin   is   used   for   creation ,  management ,  and
 deletion
```

```
        Admin   admin  =  connection . getAdmin ();
```

- Create a table

  Create a table using the specified table name. Use the default table name for MaxVersions and TimeToLive.

```
 //  Create   an   HTableDesc  riptor ,  which   contains   only
one   ColumnFami  ly
  HTableDesc  riptor   descriptor  =  new   HTableDesc  riptor (
TableName . valueOf ( TABLE_NAME ));

 //  Create   a   ColumnFami  ly .  Use   the   default
ColumnFami  ly   name   for   Max   Versions   and   TimeToLive .
The   default   ColumnFami  ly   name   for   Max   Versions   is
1   and   for   TimeToLive   is   Integer . INF_MAX
  descriptor . addFamily ( new   HColumnDes  criptor ( COLUMN_FAM
ILY_NAME ));

 //  Use   the   createTabl  e   API   of   the   Admin   to
create   a   table
  System . out . println (" Create   table  " +  descriptor .
getNameAsS  tring ());
  admin . createTabl  e ( descriptor );
```

- Write Data

  Write a row of data to Table Store.

```
 //  Create   a   Tablestore  Table   for   reading ,  writing ,
updating ,  deletion ,  and   other   operations   on   a   single
  table
  Table   table  =  connection . getTable ( TableName . valueOf (
TABLE_NAME ));

 //  Create   a   Put   object   with   the   primary   key
row_1
  System . out . println (" Write   one   row   to   the   table
");
        Put   put  =  new   Put ( ROW_KEY );

 //  Add   a   column .  Table   Store   supports   only   single
ColumnFami  lies .  The   ColumnFami  ly   name   is   configured
in   hbase - site . xml .  If   the   ColumnFami  ly   name   is
not   configured ,  the   default   name   is  " f ".  In   this
case ,  the   value   of   COLUMN_FAM  ILY_NAME   may   be   null
when   data   is   written .
  put . addColumn ( COLUMN_FAM  ILY_NAME ,  COLUMN_NAM  E ,
COLUMN_VAL  UE );

 //  Run   put   for   Table ,  and   use   HBase   APIs   to
write   the   row   of   data   to   Table   Store
        table . put ( put );
```

- Read Data

  Read data of the specified row.

```
 //  Create   a   Get   object   to   read   the   row   whose
primary   key   is   ROW_KEY .
```

```
    Result   getResult  =  table . get ( new   Get ( ROW_KEY ));
    Result   result  =  table . get ( get );

   //  Print   the   results
    String   value  =  Bytes . toString ( getResult . getValue (
  COLUMN_FAM  ILY_NAME ,  COLUMN_NAM  E ));
    System . out . println (" Get   one   row   by   row   key ");
    System . out . printf ("\ t % s  = % s \ n ",  Bytes . toString (
  ROW_KEY ),  value );
```

- **Scan data**

  Read data in the specified range.

```
   Scan   data   of   all   rows   in   the   table
   System . out . println (" Scan   for   all   rows :");
   Scan   scan  =  new   Scan ();

   ResultScan  ner   scanner  =  table . getScanner ( scan );

  //  Print   the   results   cyclically
   for  ( Result   row  :  scanner ) {
    byte []  valueBytes  =  row . getValue ( COLUMN_FAM  ILY_NAME ,
  COLUMN_NAM  E );
    System . out . println ('\ t ' +  Bytes . toString ( valueBytes
  ));
  }
```

- **Delete a table**

  Use Admin APIs to delete a table.

```
    print (" Delete   the   table ");
    admin . disableTab  le ( table . getName ());
    admin . deleteTabl  e ( table . getName ());
```

Complete code

```
package   samples ;

import   org . apache . hadoop . conf . Configurat  ion ;
import   org . apache . hadoop . hbase . HBaseConfi  guration ;
import   org . apache . hadoop . hbase . HColumnDes  criptor ;
import   org . apache . hadoop . hbase . HTableDesc  riptor ;
import   org . apache . hadoop . hbase . TableName ;
import   org . apache . hadoop . hbase . client .*;
import   org . apache . hadoop . hbase . util . Bytes ;

import   java . io . IOExceptio  n ;

public   class   HelloWorld  {

    private   static   final   byte []  TABLE_NAME  =  Bytes .
toBytes (" HelloTable  store ");
    private   static   final   byte []  ROW_KEY  =  Bytes . toBytes
(" row_1 ");
    private   static   final   byte []  COLUMN_FAM  ILY_NAME  =
Bytes . toBytes (" f ");
    private   static   final   byte []  COLUMN_NAM  E  =  Bytes .
toBytes (" col_1 ");
```

```
    private   static   final   byte []  COLUMN_VAL  UE  =  Bytes .
toBytes (" col_value ");

    public   static   void   main ( String []  args ) {
        helloWorld ();
    }

    private   static   void   helloWorld () {

        try   {
            Configurat  ion   config  =  HBaseConfi  guration .
create ();
            Connection   connection  =  Connection  Factory .
createConn  ection ( config );
            Admin   admin  =  connection . getAdmin ();

            HTableDesc  riptor   descriptor  =  new   HTableDesc
riptor ( TableName . valueOf ( TABLE_NAME ));
            descriptor . addFamily ( new   HColumnDes  criptor (
COLUMN_FAM  ILY_NAME ));

            System . out . println (" Create   table  " +  descriptor
. getNameAsS  tring ());
            admin . createTabl  e ( descriptor );

            Table   table  =  connection . getTable ( TableName .
valueOf ( TABLE_NAME ));

            System . out . println (" Write   one   row   to   the
table ");
            Put   put  =  new   Put ( ROW_KEY );
            put . addColumn ( COLUMN_FAM  ILY_NAME ,  COLUMN_NAM  E
,  COLUMN_VAL  UE );
            table . put ( put );

            Result   getResult  =  table . get ( new   Get ( ROW_KEY
));
            String   value  =  Bytes . toString ( getResult .
getValue ( COLUMN_FAM  ILY_NAME ,  COLUMN_NAM  E ));
            System . out . println (" Get   a   one   row   by   row
  key ");
            System . out . printf ("\ t % s  = % s \ n ",  Bytes .
toString ( ROW_KEY ),  value );

            Scan   scan  =  new   Scan ();

            System . out . println (" Scan   for   all   rows :");
            ResultScan  ner   scanner  =  table . getScanner ( scan
);
            for  ( Result   row  :  scanner ) {
                byte []  valueBytes  =  row . getValue ( COLUMN_FAM
ILY_NAME ,  COLUMN_NAM  E );
                System . out . println ('\ t ' +  Bytes . toString (
valueBytes ));
            }

            System . out . println (" Delete   the   table ");
            admin . disableTab  le ( table . getName ());
            admin . deleteTabl  e ( table . getName ());

            table . close ();
            admin . close ();
            connection . close ();
        }  catch  ( IOExceptio  n   e ) {
```

```
                System . err . println (" Exception   while   running
 HelloTable  store : " +  e . toString ());
                System . exit ( 1 );
         }
     }
}
```

10 Authorization management

10.1 RAM and STS

Alibaba Cloud’s permission management function include Resource Access Management (RAM) and Security Token Service (STS). This function enable users to access Table Store through RAM user accounts with different permissions, and grant users temporary access authorization.

RAM is primarily used to control account system permissions over a long-term period. It allows you to assign different permissions to different RAM users created under your primary account to implement authorization management. For more information, see RAM.

STS is a security credential (token) management system that grants temporary access permissions.

Background

RAM and STS are designed to securely grant access to users without disclosing the primary account’s AccessKey. Unintentional AccessKey disclosure poses serious account security risks as unauthorized users may freely operate the affected primary account, including malicious use of resources and theft of account information.

RAM provides permission control function used to allocate RAM users with different permissions to different entities, minimizing impact to a primary account if a RAM user’s AccessKey is disclosed. Generally, RAM users are created for long-term account operations. Therefore, the AccessKeys of RAM users must not be disclosed.

In contrast to RAM’s long-term control function, STS provides temporary access authorization by returning a temporary AccessKey and token, which can be used directly by temporary users to access Table Store. Generally, the permissions obtained from STS are more restrictive and only valid for a limited period of time.

Basic concepts

Basic concepts related to RAM and STS are described as follows:

Concept	Description
RAM user	RAM users are created under an Alibaba Cloud primary account and assigned independent passwords and permissions, with each RAM user having its own AccessKey. RAM users can be used to perform authorized operations in the same way as the primary account. Generally, RAM users can be understood as users with certain permissions or operators with permissions for specified operations.
Role	A virtual concept indicating certain operation permissions, roles do not have independent logon passwords or AccessKeys. RAM users can assume roles, and the permissions that are granted when a role is assumed belong to this role. A role may be assumed by multiple users at the same time.
Policy	Policies are rules used to define permissions, such as the permissions to read or write certain resources.
Resource	Resources are the cloud resources that users can access, such as one or all instances of Table Store, or a certain table in an instance.

The relationship between a RAM user and its roles is similar to a relationship between an individual and their social identities in different scenarios. For example, a person can assume a role of employee in a company and assume a role of parent at home. Different roles are assigned corresponding permissions. The concept of employee or parent is not an actual entity able to take actions. Roles are complete only when being assumed by RAM users. Furthermore, a role may be assumed by multiple users at the same time. The user who assumes a role is automatically assigned all permissions of the role.

The following example provides more detailed information:

Assume that an Alibaba Cloud primary account named Alice has two Table Store instances named alice_a and alice_b. Alice has full permissions on both instances.

To maintain security of the primary account, Alice uses RAM to create two RAM users: Bob and Carol. Bob has the read and write permissions on alice_a, and Carol has the read and write permissions on alice_b. Bob and Carol both have their own AccessKeys. If the AccessKey of Bob or Carol is disclosed, only the corresponding instance is affected. Alice can then cancel the permissions of the compromised RAM user through the console.

If Alice needs to authorize another RAM user to read the tables in alice_a, instead of disclosing Bob's AccessKey to the user, Alice can create a role (for example, AliceAReader), and assign the role the read permission on alice_a. However, AliceAReader cannot be used directly as no AccessKey corresponds to this role.

To obtain temporary authorization, Alice can call STS's AssumeRole interface to inform STS that RAM user Bob wants to assume the role AliceAReader. If the interface is successfully called, STS returns a temporary AccessKeyID, AccessKeySecret, and SecurityToken as the access credentials. A temporary user assigned with these credentials obtains the temporary permission to access alice_a. The credentials' expiration time is specified when the AssumeRole interface is called.

RAM and STS best practices

RAM and STS are designed with complexity to achieve flexible permission control at the cost of simplicity.

RAM users and roles are two concepts used to separate the entity that performs operations from the virtual entity that represents a permission set. If a RAM user requires many permissions (including read and write permissions) but each operation only requires part of the total permission set, you can create two roles: one with the read permission and one with the write permission. Then create a user who does not have any permissions but can assume these two roles. When a RAM user needs to read or write data, the RAM user can temporarily assume the role with the required permission. In addition, roles can be used to grant permissions to other Alibaba Cloud users, making collaborations easier while maintaining strict account security.

Implementing RAM or STS through the console and command line operations are strongly recommended to reduce the actual amount of codes that must be used. If code must be used to perform such operations, see the RAM API Reference and STS API Reference.

10.2 Customize permissions

Action

Action is an API name that is used to specify APIs that are open or restricted for user access. When creating a Table Store authorization policy, add an `ots :` prefix for each Action and separate multiple Actions using commas. The asterisk (*) wildcard is also supported (including prefix matching and suffix matching).

Typical Action

- Single API

```
" Action ": " ots : GetRow "
```

- Multiple APIs

```
" Action ": [
" ots : PutRow ",
" ots : GetRow "
]
```

- All read-only API

```
" Action ": [
" ots : BatchGet *",
" ots : Describe *",
" ots : Get *",
" ots : List *",
" ots : ComputeSpl  itPointsBy  Size "
]
```

- All read and write API

```
" Action ": " ots :*"
```

Resource

A Resource in Table Store is composed of multiple fields including product, region, user ID, instance name, and table name. Each field supports asterisk (*) wildcard (including prefix matching and suffix matching).

The format is as follows:

```
acs : ots :[ region ]:[ user_id ]: instance /[ instance_n  ame ]/
table /[ table_name ]
```

- The product is ots.
- [xxx] indicates a variable.
- The region is an abbreviation written in English, for example, cn-hangzhou. For more information about regions of service nodes, see Region.
- The user ID is the Alibaba Cloud account ID.

> Note:
> Instance names are case-insensitive. However, you must use lower case letters for `[ instance_n ame ]` in resource definition.

Typical Resource

- All resources of the users in all regions

```
" Resource ": " acs : ots :*:*:*"
```

- All instances and their tables of user 123456 in China East 1 region

```
" Resource ": " acs : ots : cn - hangzhou : 123456 : instance /*"
```

- Instance abc and its tables of user 123456 in China East 1 region

```
" Resource ": [
" acs : ots : cn - hangzhou : 123456 : instance / abc ",
" acs : ots : cn - hangzhou : 123456 : instance / abc / table /*"
]
```

- All instances whose names begin with abc and their tables

```
" Resource ": " acs : ots :*:*: instance / abc *"
```

- All instances whose names begin with abc and their tables whose names begin with xyz (excluding instance resources, and not match acs:ots:*:*:instance/abc*)

```
" Resource ": " acs : ots :*:*: instance / abc */ table / xyz *"
```

- All instances whose names end with abc and their tables whose names end with xyz.

```
" Resource ": [
" acs : ots :*:*: instance /* abc ",
" acs : ots :*:*: instance /* abc / table /* xyz "
]
```

API types

Table Store has two types of APIs

- Management APIs for reading from, and writing to, instances.
- Data APIs for reading from, and writing to, tables and rows.

The following table describes these APIS:

API/Action	API Type	Description
ListInstance	Management	Get instance list, called by console only
InsertInstance	Management	Create instance, called by console only
GetInstance	Management	Get instance meta, called by console only
DeleteInstance	Management	Delete instance, called by console only
ListTable	Data	Get table list, called by console or SDK
CreateTable	Data	Create table, called by console or SDK
UpdateTable	Data	Update table meta, called by console or SDK
DescribeTable	Data	Get table meta, called by console or SDK
DeleteTable	Data	Delete table, called by console or SDK
GetRow	Data	Read a record, called by SDK only
PutRow	Data	Insert a record, called by SDK only
UpdateRow	Data	Update a record, called by SDK only
DeleteRow	Data	Delete a record, called by SDK only

API/Action	API Type	Description
GetRange	Data	Readrange, called by SDK only
BatchGetRow	Data	Batch read records, called by SDK only
BatchWriteRow	Data	Batch write records, called by SDK only

- Resources accessed by management APIs

  Management APIs are generally instance-related operations and can be called only on the console. The actions and resources definitions of management APIs determine subsequent use of the console. The prefix `acs : ots :[ region ]:[ user_id ]:` is omitted in the following accessed resources, leaving only the instance and table parts to be described.

API/Action	Resource Access
ListInstance	instance/*
InsertInstance	instance/[instance_name]
GetInstance	instance/[instance_name]
DeleteInstance	instance/[instance_name]

- Resources accessed by data APIs

  Data APIs are genearlly table-related operations and can be called both on the console and by the SDK. The actions and resources definitions of data APIs determine subsequent use of the console. The prefix `acs : ots :[ region ]:[ user_id ]:` is omitted in the following accessed resources, leaving only the instance and table parts to be described.

API/Action	Resource Access
ListTable	instance/[instance_name]/table/*
CreateTable	instance/[instance_name]/table/[table_name]
UpdateTable	instance/[instance_name]/table/[table_name]
DescribeTable	instance/[instance_name]/table/[table_name]

API/Action	Resource Access
DeleteTable	instance/[instance_name]/table/[table_name]
GetRow	instance/[instance_name]/table/[table_name]
PutRow	instance/[instance_name]/table/[table_name]
UpdateRow	instance/[instance_name]/table/[table_name]
DeleteRow	instance/[instance_name]/table/[table_name]
GetRange	instance/[instance_name]/table/[table_name]
BatchGetRow	instance/[instance_name]/table/[table_name]
BatchWriteRow	instance/[instance_name]/table/[table_name]

- Limits
  - In a policy, actions and resources are verified by string matching. When using the asterisk (*) wildcard, prefix matching and suffix matching are distinguished. For example, if a resource is defined as `acs : ots :*:*: instance /*/`, then `acs : ots :*:*: instance / abc` cannot be matched. If a resource is defined as `acs : ots :*:*: instance / abc`, then `acs : ots :*:*: instance / abc / table / xyz` cannot be matched.
  - To manage instance resources on the Table Store console, you must have permission to read the `acs : ots :[ region ]:[ user_id ]: instance /*` resource to obtain the instance list on the console.
  - For Batch APIs (such as BatchGetRow and BatchWriteRow), the backend service performs authentication for each table being accessed. Operations can be performed only when authentication is successful for all tables. Otherwise, a permission error is returned.

Condition

The policy supports multiple authentication conditions that are supported on all APIs of Table Store, including access IP address restriction, whether to access through

HTTPS, whether to access through Multi-Factor Authentication (MFA), and access time restriction.

- Access IP address restriction

  Resource Access Management can restrict the source IP addresses used to access Table Store, and filter IP addresses based on the network segment. The following are typical application scenarios:

  - Multiple IP addresses are restricted. For example, only requests from 10.101.168.111 and 10.101.169.111 are allowed.

```
{
" Statement ": [
    {
        " Effect ": " Allow ",
        " Action ": " ots :*",
        " Resource ": " acs : ots :*:*:*",
        " Condition ": {
            " IpAddress ": {
                " acs : SourceIp ": [
                    " 10 . 101 . 168 . 111 ",
                    " 10 . 101 . 169 . 111 "
                ]
            }
        }
    }
],
" Version ": " 1 "
}
```

  - A single IP address is restricted. For example, only requests from 10.101.168.111 or 10.101.169.111/24 are allowed.

```
{
" Statement ": [
    {
        " Effect ": " Allow ",
        " Action ": " ots :*",
        " Resource ": " acs : ots :*:*:*",
        " Condition ": {
            " IpAddress ": {
                " acs : SourceIp ": [
                    " 10 . 101 . 168 . 111 ",
                    " 10 . 101 . 169 . 111 / 24 "
                ]
            }
        }
    }
],
" Version ": " 1 "
}
```

- HTTPS access restriction

  Resource Access Management can specify the use of HTTPS for access.

  Access by requests only through HTTPS

```
{
    " Statement ": [
        {
            " Effect ": " Allow ",
            " Action ": " ots :*",
            " Resource ": " acs : ots :*:*:*",
            " Condition ": {
                " Bool ": {
                    " acs : SecureTran  sport ": " true "
                }
            }
        }
    ],
    " Version ": " 1 "
}
```

- MFA access restriction

  Resource Access Management can specify the use of MFA for access.

  Access by requests only through MFA

```
{
    " Statement ": [
        {
            " Effect ": " Allow ",
            " Action ": " ots :*",
            " Resource ": " acs : ots :*:*:*",
            " Condition ": {
                " Bool ": {
                    " acs : MFAPresent  ": " true "
                }
            }
        }
    ],
    " Version ": " 1 "
}
```

- Access time restriction

  Resource Access Management can specify the time to grant access by a request, that is, it can determine if access is allowed or rejected by requests only before a specified time. For example,

  user access is allowed only before 00:00:00 January 1, 2016.

```
{
    " Statement ": [
```

```
        {
            " Effect ": " Allow ",
            " Action ": " ots :*",
            " Resource ": " acs : ots :*:*:*",
            " Condition ": {
                " DateLessTh  an ": {
                    " acs : CurrentTim  e ": " 2016 - 01 - 01T00 :
 00 : 00 + 08 : 00 "
                }
            }
        }
    ],
    " Version ": " 1 "
}
```

Typical application scenarios

This section defines specific policies in typical scenarios and offers authorization methods.

- Multiple authorization conditions

  In this scenario, users accessing the 10.101.168.111/24 network segment can read from and write to all instances named online-01 and online-02 (including all tables of these instances). A restrictive access policy means access is allowed only before 0:00:00 January 1, 2016 through HTTPS.

  To grant policy permissions to a RAM user, follow these steps:

  1. Use the primary account to log on to the RAM console.
  2. In the left-side navigation pane, click Policies.
  3. In the upper-right corner, click Create Authorization Policy.
  4. Select Blank Template.
  5. Enter the Authorization Policy Name and copy the following content to Policy Content.

  ```
  {
  " Statement ": [
      {
          " Effect ": " Allow ",
          " Action ": " ots :*",
          " Resource ": [
              " acs : ots :*:*: instance / online - 01 ",
              " acs : ots :*:*: instance / online - 01 / table /*",
              " acs : ots :*:*: instance / online - 02 ",
              " acs : ots :*:*: instance / online - 02 / table /*"
          ],
          " Condition ": {
              " IpAddress ": {
                  " acs : SourceIp ": [
                      " 10 . 101 . 168 . 111 / 24 "
                  ]
  ```

```
            },
            " DateLessTh  an ": {
                " acs : CurrentTim  e ": " 2016 - 01 - 01T00 : 00
 : 00 + 08 : 00 "
            },
            " Bool ": {
                " acs : SecureTran  sport ": " true "
            }
        }
    }
],
" Version ": " 1 "
}
```

6. Click Create Authorization Policy and then click Close.
7. In the left-side navigation pane, click Users.
8. Locate the RAM user to be authorized, and click Authorize.
9. Select the policy created in the preceding steps.
10. Click OK.

- Reject requests

  In this scenario, users accessing the IP address 10.101.169.111 are not allowed to write to all tables of instances in the Beijing region whose names begin with `online` and `product`. Operations related to instances are not involved.

  To reject requests, first see the preceding steps to create a new policy and grant policy permissions to the designated RAM user. Then, during policy creation, copy the following content to Policy Content.

```
{
    " Statement ": [
        {
            " Effect ": " Deny ",
            " Action ": [
                " ots : Create *",
                " ots : Insert *",
                " ots : Put *",
                " ots : Update *",
                " ots : Delete *",
                " ots : BatchWrite *"
            ],
            " Resource ": [
                " acs : ots : cn - beijing :*: instance / online */
 table /*",
                " acs : ots : cn - beijing :*: instance / product */
 table /*"
            ],
            " Condition ": {
                " IpAddress ": {
                    " acs : SourceIp ": [
                        " 10 . 101 . 169 . 111 "
                    ]
                }
            }
```

```
        }
    ],
    " Version ": " 1 "
}
```