ALIBABA CLOUD

Alibaba Cloud

Tablestore
Function Introduction

Document Version: 20220711

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, 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 this document 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 directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
⚠ Danger	A danger notice 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.
🔔 Warning	A warning notice 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 restart an instance.
📢 Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	📢 **Notice:** If the weight is set to 0, the server no longer receives new requests.
? Note	A note indicates supplemental instructions, best practices, tips, and other content.	? **Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings> Network> Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Overview

Tablestore is a multi-model service that is developed by Alibaba Cloud. Tablestore can store a large amount of structured data and supports fast query and analysis. The distributed storage and powerful index-based search engine allow Tablestore to store petabytes of data while transactions per second (TPS) of tens of millions and a latency within milliseconds are ensured.

Terms

Before you use Tablestore, you must understand the terms described in the following table.

Term	Description
Instance	An entity that uses and manages Tablestore. Tablestore implements access control and resource metering for applications at the instance level.
Read/write throughput	The read/write throughput is measured by read/write capacity units (CUs), which is the smallest billing unit for read and write operations.
Region	A region is a physical data center of Alibaba Cloud.
Endpoint	Each Tablestore instance has an endpoint. An endpoint must be specified before operations can be performed on tables or data in Tablestore.

Models

Tablestore provides multiple models that you can select and apply to your business. The following table describes the models of Tablestore.

Model	Description
Wide Column model	The Wide Column model is applicable to various scenarios such as metadata and big data. This model supports multiple features such as data versions, time to live (TTL), auto-increment of primary key columns, conditional updates, atomic counters, and filters.
Timeline model	The Timeline model is a data model that is suitable for scenarios such as instant messaging (IM) and feed streams. This model can also meet 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 search and Boolean query.

Features

The following table describes the features provided by Tablestore.

Feature	Description
Auto-increment of primary key columns	After you set a primary key column (non-partition key) to an auto-increment column, you do not need to enter values in this column when you write data in a row. Instead, Tablestore automatically generates primary key column values. The automatically generated values are unique within the rows that share the same partition key. These values increase sequentially.

Feature	Description
Conditional update	If you use conditional update, data in the table can be updated only when the conditions are met. If the conditions are not met, the update fails.
Atomic counters	Columns are used as an atomic counter. The atomic counter provides real-time statistics for some online applications, such as calculating the real-time page views (PVs) of a post.
Configure a filter	Filters can be used to sort results on the server side. Only results that match the filter conditions are returned. The feature effectively reduces the volume of transferred data and shortens the response time.
Search index	Based on inverted index and column-oriented storage, search index solves the complex query problem in big data scenarios.
Secondary index	Tablestore allows you to create a secondary index on an attribute column.
Tunnel Service	Tunnel Service provides tunnels that are used to export and consume distributed data in full, incremental, and differential modes in real time. After tunnels are created, you can consume historical and incremental data exported from a specified table.
HBase support	Tablestore HBase Client can be used to access Tablestore by using Java applications built on open source HBase APIs.

2.Features and regions

This topic describes the features that are in invitational preview and the features that are supported only in specific regions.

Features in invitational preview and supported regions

The local transaction feature is in invitational preview and are supported in all regions. You must submit a ticket to enable this feature.

Features supported in specific regions

The search index, Tunnel Service, TimeSeries model, SQL query, and data delivery features are supported only in specific regions. The following table describes the features that are supported only in specific regions.

Note In the following table, a tick (√) indicates that the feature is supported in this region, and a cross (×) indicates that the feature is not supported in this region.

Region	Search index	Tunnel Service	TimeSeries model	SQL query	Data delivery
China (Hangzhou)	√	√	√	√	√
China East 1 Finance	√	√	×	×	×
China (Shanghai)	√	√	√	√	√
China East 2 Finance	×	√	×	×	×
China (Qingdao)	×	×	×	×	×
China (Beijing)	√	√	×	√	√
China (Zhangjiakou)	√	√	×	√	√
China (Hohhot)	×	×	×	×	×
China (Shenzhen)	√	√	√	√	×
China (Chengdu)	×	×	×	×	×
China (Hong Kong)	√	√	×	×	×

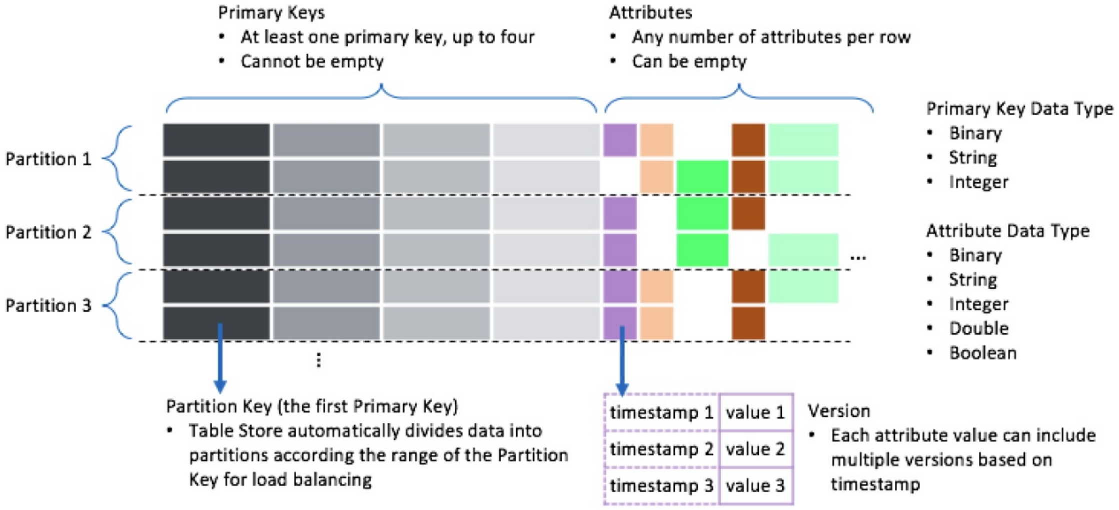
Region	Search index	Tunnel Service	TimeSeries model	SQL query	Data delivery
Singapore	√	√	×	√	×
Australia (Sydney)	√	√	×	×	×
Malaysia (Kuala Lumpur)	×	×	×	×	×
Indonesia (Jakarta)	√	√	×	×	×
Japan (Tokyo)	√	√	×	×	×
Germany (Frankfurt)	√	√	√	√	×
UK (London)	√	√	×	×	×
US (Silicon Valley)	√	√	×	×	×
US (Virginia)	√	√	×	×	×
India (Mumbai)	√	√	×	×	×
UAE (Dubai)	×	×	×	×	×
Philippines (Manila)	√	√	×	×	×

3.Wide Column model

3.1. Overview

Tablestore is a multi-model service developed by Alibaba Cloud. Tablestore can store a large amount of structured data. Wide Column is one of the models used in Tablestore. This topic describes the components of the Wide Column model and the differences between the Wide Column model and relational model.

Components


Primary Keys
• At least one primary key, up to four
• Cannot be empty
Attributes
• Any number of attributes per row
• Can be empty
Partition 1
Partition 2
Partition 3
...
⋮
Primary Key Data Type
• Binary
• String
• Integer
Attribute Data Type
• Binary
• String
• Integer
• Double
• Boolean
Partition Key (the first Primary Key)
• Table Store automatically divides data into partitions according the range of the Partition Key for load balancing
timestamp 1
value 1
timestamp 2
value 2
timestamp 3
value 3
Version
• Each attribute value can include multiple versions based on timestamp

The preceding figure shows the components of the Wide Column mode. The following table lists the components of the Wide Column model.

Components	Description
Primary key	Primary keys uniquely identify each row in tables. A primary key consists of one to four primary key columns.
Partition key	The first primary key column is called the partition key. Tablestore partitions data in a table based on partition keys. Rows that share the same partition key values are distributed to the same partition for load balancing.
Attribute column	All columns except for the primary key columns in a row are called attributed columns. Each attribute column can contain values of different versions. Tablestore does not impose limits on the number of attribute columns that can be contained in each row.
Version	Each value in an attribute column has a unique version number. The version number uses a timestamp based on which to define time to live (TTL).
Data type	Tablestore supports the following data types: STRING, BINARY, DOUBLE, INTEGER, and BOOLEAN.
TTL	You can configure TTL for each table. If you set TTL to one month for a table, Tablestore deletes data that is stored in the table from the previous month.

Components	Description
Max versions	You can set the maximum number of versions for the value in each attribute column of one table. Max versions can be used to control the number of versions for the value in each attribute column. When the actual number of versions in an attribute column exceeds the max versions value, Tablestore asynchronously deletes earlier versions.

Compare the Wide Column model with the relational model

The following table lists the differences between the Wide Column model and relational model.

Model	Feature
Wide Column model	Three-dimensional structure (row, column, and time), schema-free data, wide row, max versions, and TTL.
Relational model	Two-dimensional structure (row and column) and traditional schema.

3.2. Primary keys and attributes

In Tablestore, tables, rows, primary keys, and attributes are the core components. A table is a collection of rows. Each row consists of a primary key and attribute. The first primary key column is called the partition key.

Primary keys

Primary keys uniquely identify each row in tables. A primary key consists of one to four primary key columns. When you create a table, you must specify primary key columns, including the name, data type, and sequence of the primary key columns.

Tablestore indexes data of a table based on the primary key columns of the table.

Partition keys

The first primary key column is called the partition key. Tablestore distributes each row of data to the corresponding partitions based on the range of the partition key values for load balancing. Rows that share the same partition key values belong to the same partition. A partition may store rows that have multiple partition key values. Tablestore splits and merges partitions based on specified rules.

> **Note** Partition keys are the basic unit to partition data. Data that shares the same partition key value cannot be split. To prevent partitions from being too large to split, we recommend that the total size of all rows that share the same partition key value is at most 10 GB.

Attributes

A row consists of multiple attribute columns. Tablestore does not impose limits on the number of attribute columns that can be contained in each row. Each row can contain a different number of attribute columns. The value in an attribute column of a row can be null. The values in an attribute column of multiple rows can be of different data types.

Attribute columns support the version feature. Multiple versions of a value in an attribute column can be retained for queries and use. You can configure time to live (TTL) for values in attribute columns. For more information, see Max versions and TTL.

3.3. Data versions and TTL

You can use data versions and time to live (TTL) to manage data in an efficient manner and minimize storage usage and storage costs.

Version number

When you update a value in an attribute column, a new version is generated for the value. The version value uses a timestamp as the version number.

When you write data to an attribute column, you can specify a version number. If you do not specify a version number, Tablestore automatically generates a version number. Version numbers are timestamps in milliseconds. When you compare version numbers with TTL values or when you calculate the values of max version offset, you must divide the version numbers by 1000.

- By default, the version number that is generated by Tablestore is the number of milliseconds that have elapsed since January 1, 1970, 00:00:00 UTC.
- If you specify a version number, make sure that the version number of the attribute column is a 64-bit timestamp that is accurate to milliseconds and within the valid version range.

You can use version numbers to implement the following features:

- TTL

  You can specify the TTL of attribute column values in a data table based on version numbers. When data of a specific version in an attribute column is retained for a period of time that exceeds the TTL value, Tablestore automatically deletes the data of the specific version.

  For example, the version number of the data in an attribute column is 1468944000000, which is equivalent to 00:00:00 UTC on July 20, 2016. If you set the value of TTL to 86400 seconds (one day), the data of this version expires at 00:00:00 UTC on July 21, 2016. Tablestore automatically deletes the data after the data expires.

- Maximum number of versions or range of version numbers that you want Tablestore to read from each attribute column

  When Tablestore reads a row of data, you can specify the maximum number of versions or the range of version numbers that you want Tablestore to read from each attribute column.

Max versions

Max versions specifies the maximum number of versions that can be retained for the data in an attribute column. If the number of versions of data in attribute columns exceeds the value of this parameter, the system deletes the data of earlier versions.

When you create a data table, the default value 1 is used if you do not specify a value for max versions. You can also specify a max versions value for data in attribute columns. After you create a data table, you can call the UpdateTable operation to modify the max versions value for the data table.

If the number of versions exceeds the max versions value that you specify, the data of earlier versions becomes invalid and you cannot read the data even if the data has not been deleted by Tablestore.

- When you decrease the value of max versions, Tablestore asynchronously deletes data of earlier versions if the number of data versions exceeds the most recent value of max versions.

- When you increase the value of max versions, you can read the data of specific versions if Tablestore has not deleted the data of the versions and the versions are within the most recent valid version range.

TTL

TTL specifies the validity period of data in the Tablestore data table in seconds. When data of a specific version in an attribute column is retained for a period of time that exceeds the TTL value, Tablestore automatically deletes the data of the specific version from the attribute column. If data in all attribute columns of a row is retained for a period of time that exceeds the TTL value, Tablestore automatically deletes the row.

For example, you set the TTL of the data in a data table to 86400 seconds (one day). At 00:00:00 UTC on July 21, 2016, data whose version number is smaller than 1468944000000 expires. This version number is equivalent to 00:00:00 UTC on July 20, 2016 after the version number is divided by 1000 to convert into seconds. Tablestore automatically deletes the expired data.

When you create a data table, the default value -1 is used if you do not specify a value for TTL. The value of -1 indicates that data in the data table never expires. You can also specify a TTL value. After you create a data table, you can call the UpdateTable operation to modify the TTL value for the data in the data table.

If data is retained for a period of time that exceeds the TTL value, the data becomes invalid and you cannot read the data even if the data has not been deleted by Tablestore.

- When you decrease the TTL value, Tablestore asynchronously deletes the data that is retained for a period of time that exceeds the most recent TTL value.
- When you increase the TTL value, you can read the data of specific versions if Tablestore has not deleted the data and data of the versions is retained for a period of time that is within the most recent TTL value.

Max version offset

If the difference between the current time and the version timestamp that you specify exceeds the TTL value that you specified for the data table, the written data immediately expires. To resolve this issue, Tablestore allows you to configure the max version offset feature.

Max version offset specifies the maximum difference between the current system time and the specified version number in seconds. The value of max version offset is a positive integer that can be greater than the number of seconds that have elapsed since 00:00:00 UTC on January 1, 1970.

When you write data to an attribute column, Tablestore checks the version number of the data. The valid version range of data in an attribute column is calculated by using the following formula: Valid version range = **[max{Data written time - Max version offset, Data written time - TTL value}, Data written time + Max version offset)**. You can write data to the attribute column only if the version number of the data (converted into seconds by dividing by 1000) is within the valid version range.

For example, you set the max version offset of the data in each attribute column of a data table to 86400 seconds (one day). At 00:00:00 UTC on July 21, 2016, only the data whose version range is between 1468944000000 (00:00:00 UTC on July 20, 2016) and 1469116800000 (00:00:00 UTC on July 22, 2016) can be written. If you attempt to write a row of data in which the version number of data in an attribute column is 1468943999000 (23:59:59 UTC on July 19, 2016), the row fails to be written.

When you create a data table, the default value 86400 is used if you do not specify a value for max version offset. You can also specify a max version offset value for the data in the data table. After you create a data table, you can call the UpdateTable operation to modify the max version offset value for the data in the data table.

3.4. Naming conventions and data types

This topic describes the naming conventions for tables and columns, as well as the data types supported by primary key columns and attribute columns.

Naming conventions

The following table describes the naming conventions for tables and columns.

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 in the same instance. • Table names can be the same in different instances.

Data types

Data types supported by primary key columns include STRING, INTEGER, and BINARY. Data types supported by attribute columns include STRING, INTEGER, DOUBLE, BOOLEAN, and BINARY.

- Data types supported by primary key columns

Data type	Definition	Size limit
String	Data is UTF-8 encoded. Empty strings are allowed.	Up to 1 KB
Integer	Data is 64-bit integers. Auto-increment columns are supported.	8 Bytes
Binary	Data is of the BINARY type. Empty values are allowed.	Up to 1 KB in length

- Data types supported by attribute columns

Data type	Definition	Size limit
String	Data is UTF-8 encoded. Empty strings are allowed.	For more information, see General limits.
Integer	Data is 64-bit integers.	8 Bytes
Double	Data is 64-bit and of the DOUBLE type.	8 Bytes
Boolean	Data is of the BOOLEAN type. The value can be True or False.	1 Byte
Binary	Data is of the BINARY type. Empty values are allowed.	For more information, see General limits.

3.5. Basic operations on data

3.5.1. Single-row operations

Tablestore provides the following single-row operations: PutRow, GetRow, UpdateRow, and DeleteRow.

> **Note** Rows are the basic units of tables. Rows consist of primary keys and attributes. A primary key is required for each row. Rows within a table contain primary key columns of the same names and same data types. Attributes are optional for each row. Rows within a table can contain different attributes. For more information, see Overview.

API operations

Single-row operations include PutRow, GetRow, UpdateRow, and DeleteRow. The following table describes these operations.

API operation	Description
GetRow	Reads a row.
PutRow	Inserts a row into a table. If the row exists, data in the row is deleted and new data is inserted.
UpdateRow	Updates a row. You can add attribute columns to or delete attribute columns from a row. You can also update the values of attribute columns in a row. If the row does not exist, a new row is added. However, if an UpdateRow operation only specifies columns to delete from a row and the row does not exist, no row is inserted into the table.
DeleteRow	Deletes a row. If the row that you want to delete does not exist, the table remains unchanged.

Use Tablestore SDKs

You can use the following Tablestore SDKs to perform single-row operations:

- Tablestore SDK for Java: Single-row operations
- Tablestore SDK for Go: Single-row operations
- Tablestore SDK for Python: Single-row operations
- Tablestore SDK for Node.js: Single-row operations
- Tablestore SDK for .NET: Single-row operations
- Tablestore SDK for PHP: Single-row operations

PutRow

You can call this operation to insert a row. If the row exists, the PutRow operation deletes all versions of data in all columns from the existing row and then inserts a new row.

CU consumption

The number of read and write capacity units (CUs) consumed by a PutRow operation is calculated based on the following rules:

- The number of consumed write CUs is rounded up from the calculation results of the following formula: [(Size of the data in all the primary key columns of the row + Size of the data in the inserted attribute columns)/4 KB].
- If the value of the condition field is not IGNORE, the PutRow operation consumes read CUs. The number of consumed read CUs is rounded up from the calculation results of the following formula: Number of consumed read CUs = Size of the data in all primary key columns of the row/4 KB.
- If row existence conditions are not met, the operation fails and one write CU and one read CU are consumed.

Operation result description

Responses vary based on whether an operation succeeds.

- If an operation succeeds, Tablestore returns the number of CUs consumed by the operation.

  Note Write operations consume read CUs based on the conditions that you specify.

  By specifying the condition field in the write request of a single row, you can specify whether to perform row existence check before the write operation. The following table describes valid values of the condition field.

condition	Description
IGNORE	No row existence check is performed.
EXPECT_EXIST	The row is expected to exist. ○ If the row exists, the operation succeeds. ○ If the row does not exist, the operation fails.

condition	Description
EXPECT_NOT_EXIST	The row is not expected to exist. ○ If the row does not exist, the operation succeeds. ○ If the row exists, the operation fails. ⓘ **Note** If you set condition to EXPECT_NOT_EXIST in a DeleteRow or UpdateRow operation, the operation is invalid because rows that do not exist cannot be deleted or updated. To update a row that does not exist, you can use the PutRow operation.

- If errors occur, for example, a parameter fails to be checked, excessive data exists in a row, or a row existence check fails, Tablestore returns the specific error codes.

Parameters

Parameter	Description
tableName	The name of the data table.
primaryKey	The primary key of the row. ⓘ **Note** • The number and types of the primary key columns that you specify must be the same as the actual number and types of primary key columns in the data table. • If a primary key column is an auto-increment primary key column, you need to only set the value of the auto-increment primary key column to a placeholder. For more information, see Auto-increment of primary key columns.
condition	The condition that you can configure to perform the PutRow operation. You can configure a row existence condition or a condition based on column values. For more information, see Conditional update. ⓘ **Note** • RowExistenceExpectation.IGNORE indicates that new data is inserted into a row regardless of whether the specified row exists or not. If the specified row exists, the existing data is overwritten. • RowExistenceExpectation.EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten. • RowExistenceExpectation.EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist.

Parameter	Description
column	The attribute column of the row. • An attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType, which is optional), and timestamp (optional). • You can set ColumnType to ColumnType.INTEGER, ColumnType.STRING, ColumnType.BINARY, ColumnType.BOOLEAN, or ColumnType.DOUBLE, which separately indicates INTEGER, STRING (a UTF-8 encoded string), BINARY, BOOLEAN, or DOUBLE. If you want to set the column value type to BINARY, you must set ColumnType to ColumnType.BINARY. If you want to use other types of column values, the setting of ColumnType is optional. • The timestamp is the data version number. For more information, see Data versions and TTL. You can specify a data version number or use the data version number generated by Tablestore. If you do not specify this parameter, the data version number generated by Tablestore is used. ○ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ○ If you specify the version number, make sure that the version number is a 64-bit timestamp accurate to milliseconds within the valid version range.

Examples

-
-
-
-

GetRow

You can call this operation to read a row.

CU consumption

The GetRow operation does not consume write CUs. The number of consumed read CUs is rounded up from the calculation result of the following formula: Number of consumed read CUs = [(Size of the data in all the primary key columns of the row + Size of the data in the attribute columns that are actually read)/4 KB]. If the specified row does not exist, one read CU is consumed by the operation.

Operation result description

One of the following results of the read request may be returned:

- If the row exists, the primary key columns and attribute columns of the row are returned.
- If the row does not exist, no row is returned and no error is reported.

Parameters

Parameter	Description
tableName	The name of the data table.
primaryKey	The primary key of the row. Note The number and types of the primary key columns that you specify must be the same as the actual number and types of primary key columns in the data table.
columnsToGet	The columns that you want to read. You can specify the names of primary key columns or attribute columns. If you do not specify a column name, all data in the row is returned. Note - By default, Tablestore returns the data from all columns of the row when you query a row. You can use the columnsToGet parameter to return specific columns. For example, if col0 and col1 are added to columnsToGet, only the values of the col0 and col1 columns are returned. - If you configure columnsToGet and filter at the same time, Tablestore first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.
maxVersions	The maximum number of data versions that you want to read. Note You must configure at least one of the following parameters: maxVersions and timeRange. - If only maxVersions is specified, data of up to the maximum number of versions is returned from the latest to the earliest. - If only timeRange is specified, all data whose versions are within the specified time range or data of the specified versions is returned. - If both maxVersions and timeRange are specified, data of up to the maximum number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
timeRange	The range of versions or specific versions that you want to read. For more information, see TimeRange. Note You must configure at least one of the following parameters: maxVersions and timeRange. - If only maxVersions is specified, data of up to the maximum number of versions is returned from the latest to the earliest. - If only timeRange is specified, all data whose versions are within the specified time range or data of the specified versions is returned. - If both maxVersions and timeRange are specified, data of up to the maximum number of versions within the time range is returned from the latest to the earliest. - To query data within a range, you must configure start and end. start indicates the start timestamp. end indicates the end timestamp. The specified range is a left-closed and right-open interval. - To query data of a specific version, you must configure timestamp. timestamp indicates a specified timestamp. You need to only configure one of timestamp and [start, end). The timestamp value ranges from 0 to Long.MAX_VALUE. Unit: milliseconds.
filter	The filter used to filter the query results on the server side. Only rows that meet the filter conditions are returned. For more information, see Configure a filter. Note If you configure columnsToGet and filter at the same time, Tablestore first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.

Examples

-
-
-

UpdateRow

You can call this operation to update the data of a specified row. You can add attribute columns to or delete attribute columns from a row, delete a specified version of data from an attribute column, or update the existing data in an attribute column. If the row does not exist, a new row is added.

> **Note** If an UpdateRow operation only specifies columns to delete from a row and the row does not exist, no row is inserted into the table.

CU consumption

The number of read and write capacity units (CUs) consumed by a UpdateRow operation is calculated based on the following rules:

- The number of consumed write CUs is rounded up from the calculation results of the following formula: [(Size of the data in all the primary key columns of the row + Size of the data in the updated attribute columns)/4 KB].

  If the UpdateRow request contains deletion instructions on specified attribute columns, the length of the name of each attribute column to delete is calculated as the column size.

- If the value of the condition field is not IGNORE, the UpdateRow operation consumes read CUs. The number of consumed read CUs is rounded up from the calculation results of the following formula: Number of consumed read CUs = Size of the data in all primary key columns of the row/4 KB.
- If row existence conditions are not met, the operation fails and one write CU and one read CU are consumed.

Parameters

Parameter	Description
tableName	The name of the data table.
primaryKey	The primary key of the row. **Note** The number and types of the primary key columns that you specify must be the same as the actual number and types of primary key columns in the data table.
condition	The condition that you can configure to perform the UpdateRow operation. You can configure a row existence condition or a condition based on column values. For more information, see Conditional update.

Parameter	Description
column	The attribute column you want to update. ● An attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value, attribute column value type (optional), and timestamp (optional). A timestamp is the data version number. You can specify a data version number or use the data version number generated by Tablestore. By default, if you do not specify this parameter, the data version number generated by Tablestore is used. For more information, see Data versions and TTL. ○ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ○ If you specify the version number, make sure that the version number is a 64-bit timestamp accurate to milliseconds within the valid version range. ● To delete a specified version of data from an attribute column, you need to only set the attribute column name and timestamp. The timestamp is a 64-bit integer that indicates a specified version of data. Unit: milliseconds. ● To delete an attribute column, you need to only set the attribute column name. **Note** A row exists even if all attribute columns in the row are deleted. To delete a row, use the DeleteRow operation.

Examples

-
-

DeleteRow

You can call this operation to delete a row. If the row to delete does not exist, the table remains unchanged.

CU consumption

The number of read and write CUs consumed by a DeleteRow operation is calculated based on the following rules:

- The number of consumed write CUs is rounded up from the calculation result of the following formula: Number of consumed write CUs = Size of the data in all primary key columns of the row/4 KB.
- If the value of the condition field is not IGNORE, the DeleteRow operation consumes read CUs. The number of consumed read CUs is rounded up from the calculation results of the following formula: Number of consumed read CUs = Size of the data in all primary key columns of the row/4 KB.
- If row existence conditions are not met, the operation fails and one write CU is consumed.

Parameters

Parameter	Description
tableName	The name of the data table.
primaryKey	The primary key of the row. Note The number and types of the primary key columns that you specify must be the same as the actual number and types of primary key columns in the data table.
condition	The condition that you can configure to perform the DeleteRow operation. You can configure a row existence condition or a condition based on column values. For more information, see Conditional update.

Examples

-
-

3.5.2. Multi-row operations

Tablestore provides multi-row operations such as BatchWriteRow, BatchGetRow, and GetRange.

> **Note** Rows are the basic units of tables. Rows consist of primary keys and attributes. A primary key is required for each row. Rows within a table contain primary key columns of the same names and same data types. Attributes are optional for each row. Rows within a table can contain different attributes. For more information, see Overview.

API operations

Multi-row operations include BatchWriteRow, BatchGetRow, and GetRange. The following table describes the operations.

API operation	Description
BatchGetRow	Reads multiple rows of data from one or more tables.
BatchWriteRow	Inserts rows into, deletes rows from, or updates rows in one or more tables.
GetRange	Queries data whose primary key is within a specified range.

Use Tablestore SDKs

You can use the following Tablestore SDKs to perform multi-row operations:

- Tablestore SDK for Java: Multi-row operations
- Tablestore SDK for Go: Multi-row operations
- Tablestore SDK for Python: Multi-row operations
- Tablestore SDK for Node.js: Multi-row operations
- Tablestore SDK for .NET: Multi-row operations

- Tablestore SDK for PHP: Multi-row operations

BatchWriteRow

You can call this operation to write multiple rows to one or more tables in a request. The BatchWriteRow operation is a set of PutRow, UpdateRow, or DeleteRow operations. When you call the BatchWriteRow operation, the process of constructing the PutRow, UpdateRow, or DeleteRow operations is the same as the process of constructing the PutRow, UpdateRow, or DeleteRow operation when you call the PutRow, UpdateRow, or DeleteRow operation. BatchWriteRow supports conditional updates.

If you call the BatchWriteRow operation, each PutRow, UpdateRow, or DeleteRow operation is separately performed. The response to each PutRow, UpdateRow, or DeleteRow operation is separately returned.

When you call the BatchWriteRow operation to write multiple rows at a time, some rows may fail to be written. If this happens, Tablestore does not return exceptions, but return BatchWriteRowResponse in which the indexes and error messages of the failed rows are included. Therefore, when you call the BatchWriteRow operation, you must check the return values. You can use the isAllSucceed method of BatchWriteRowResponse to check whether all rows are written. If you do not check the return values, you may ignore the rows that fail to be written.

If the server detects that invalid parameters exist in some operations, the BatchWriteRow operation may return an exception about parameter errors before the first operation in the request is performed.

Parameters

For more information, see Single-row operations.

Examples

BatchGetRow

You can call this operation to read multiple rows from one or more tables in a request. The BatchGetRow operation is a set of GetRow operations. When you call the BatchGetRow operation, the process of constructing the GetRow operations is the same as the process of constructing the GetRow operation when you call the GetRow operation.

Note that the BatchGetRow operation uses the same parameter configurations for all rows. For example, if ColumnsToGet is set to [colA], only the value of the colA column is read from all rows.

If you call the BatchGetRow operation, each GetRow operation is separately performed and the response to each GetRow operation is separately returned.

When you call the BatchGetRow operation to read multiple rows at a time, some rows may fail to be read. If this happens, Tablestore does not return exceptions, but return BatchGetRowResponse in which error messages of the failed rows are included. Therefore, when you call the BatchGetRow operation, you must check the return values. You can use the isAllSucceed method of BatchGetRowResponse to check whether all rows are read or use the getFailedRows method of BatchGetRowResponse to obtain the information about failed rows.

Parameters

For more information, see Single-row operations.

Examples

GetRange

You can call this operation to read data whose primary key is within a specified range. The primary key range is a left-closed and right-open interval.

The GetRange operation allows you to read data whose primary key is within a specified range in a forward or backward direction. You can also specify the number of rows to read. If the range is large and the number of scanned rows or the volume of scanned data exceeds the upper limit, the scan stops, and the rows that are read and information about the primary key of the next row are returned. You can initiate a request to start from where the last operation left off and read the remaining rows based on information about the primary key of the next row returned by the previous operation.

> **Note** In Tablestore tables, all rows are sorted by primary key. The primary key of a table sequentially consists of all primary key columns. Therefore, do not assume that the rows are sorted based on a specific primary key column.

Usage notes

GetRange follows the leftmost matching principle. Tablestore compares values in sequence from the first primary key column to the last primary key column to read data whose primary key is within a specified range. For example, the primary key of a data table consists of the following primary key columns: PK1, PK2, and PK3. When data is read, Tablestore first determines whether the PK1 value of a row is within the range that is specified for the first primary key column. If the PK1 value of a row is within the range, Tablestore stops determining whether the values of other primary key columns of the row are within the ranges that are specified for each primary key column and returns the row. If the PK1 value of a row is not within the range, Tablestore continues to determine whether the values of other primary key columns of the row are within the ranges that are specified for each primary key column in the same manner as PK1. For more information about range query principles, see Detailed explanation of GetRange.

If one of the following conditions is met, the GetRange operation may stop and return data:

- The amount of scanned data reaches 4 MB.
- The number of scanned rows reaches 5,000.
- The number of returned rows reaches the limit.
- The read throughput is insufficient to read the next row of data because all reserved read throughput is consumed.

CU consumption

The number of read CUs consumed by a GetRange operation is calculated from the start point of the range to the start point of the next row that is unread. The number of read CUs that are consumed for the GetRange operation is rounded up from the calculation result of the following formula: Number of consumed read CUs = (Size of the data in all primary key columns of the rows that meet the query conditions + Size of the data in the attribute columns that are read)/4 KB. For example, if 10 rows that meet the query conditions are read and the sum of the size of the data in all primary key columns of the rows and the size of the data in the attribute columns that are read is 330 bytes, the number of consumed read CUs is rounded up from the calculation result of the following formula: Number of consumed read CUs = (3.3 KB/4 KB). In this case, the GetRange operation consumes one read CU.

Parameters

Parameter	Description
tableName	The name of the data table.
direction	The order in which you want to sort the rows in the response. - If you set this parameter to FORWARD, the value of the inclusiveStartPrimaryKey parameter must be smaller than the value of the exclusiveEndPrimaryKey parameter, and the rows in the response are sorted in ascending order of primary key values. - If you set of this parameter to BACKWARD, the value of the inclusiveStartPrimaryKey parameter must be greater than the value of the exclusiveEndPrimaryKey parameter, and the rows in the response are sorted in descending order of primary key values. For example, if you set the direction parameter to FORWARD for a table that contains two primary keys A and B and the value of A is smaller than the value of B, the rows whose primary key values are greater than or equal to the value of A but smaller than the value of B are returned in ascending order from A to B. If you set the direction parameter to BACKWARD, the rows whose primary key values are smaller than or equal to the value of B and greater than the value of A are returned in descending order from B to A.
inclusiveStartPrimaryKey	The start and end primary keys of the range to read. The start and end primary keys must be valid primary keys or virtual points that consist of the INF_MIN and INF_MAX type data. The number of columns for each virtual point must be the same as the number of columns of each primary key. INF_MIN indicates an infinitely small value. All values of other types are greater than the INF_MIN type value. INF_MAX indicates an infinitely great value. All values of other types are smaller than the INF_MAX type value. - inclusiveStartPrimaryKey indicates the start primary key. If the row that contains the start primary key exists, the row of data is returned. - exclusiveEndPrimaryKey indicates the end primary key. No matter whether the row that contains the end primary key exists, the row of data is not returned. The rows in the data table are sorted in ascending order based on the primary key values. The range to read is a left-closed and right-open interval. If data is read in the forward direction, the rows whose primary keys are greater than or equal to the start primary key but smaller than the end primary key are returned.
exclusiveEndPrimaryKey	
limit	The maximum number of rows that you want to return. The value of this parameter must be greater than 0. An operation stops after the maximum number of rows that you want to return in the forward or backward direction is reached, even if some rows within the specified range are not returned. You can use the value of nextStartPrimaryKey returned in the response to read remaining data in the next request.

Parameter	Description
columnsToGet	The columns that you want to return. You can specify the names of primary key columns or attribute columns. If you do not specify a column name, all data in the row is returned. Note • By default, Tablestore returns the data from all columns of the row when you query a row. You can use the columnsToGet parameter to return specific columns. For example, if col0 and col1 are added to columnsToGet, only the values of the col0 and col1 columns are returned. • If a row is within the specified range to be read based on the primary key value but does not contain the specified columns to return, the response excludes the row. • If you configure columnsToGet and filter at the same time, Tablestore first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.
maxVersions	The maximum number of data versions that you want to read. Note You must configure at least one of the following parameters: maxVersions and timeRange. • If only maxVersions is specified, data of up to the maximum number of versions is returned from the latest to the earliest. • If only timeRange is specified, all data whose versions are within the specified time range or data of the specified versions is returned. • If both maxVersions and timeRange are specified, data of up to the maximum number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
timeRange	The range of versions or specific versions that you want to read. For more information, see TimeRange. **Note** You must configure at least one of the following parameters: maxVersions and timeRange. - If only maxVersions is specified, data of up to the maximum number of versions is returned from the latest to the earliest. - If only timeRange is specified, all data whose versions are within the specified time range or data of the specified versions is returned. - If both maxVersions and timeRange are specified, data of up to the maximum number of versions within the time range is returned from the latest to the earliest. - To query data within a range, you must configure start and end. start indicates the start timestamp. end indicates the end timestamp. The specified range is a left-closed and right-open interval, which includes the start value and excludes the end value. - To query data of a specific version, you must configure timestamp. timestamp indicates a specified timestamp. You need to only configure one of timestamp and [start, end). The timestamp value ranges from 0 to Long.MAX_VALUE. Unit: milliseconds.
filter	The filter used to filter the query results on the server side. Only rows that meet the filter conditions are returned. For more information, see Configure a filter. **Note** If you configure columnsToGet and filter at the same time, Tablestore first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.
nextStartPrimaryKey	The start primary key of the next read request. The value of nextStartPrimaryKey can be used to determine whether all data is read. - If the value of nextStartPrimaryKey is not empty in the response, the nextStartPrimaryKey value can be used as the value of the start primary key for the next GetRange operation. - If the value of nextStartPrimaryKey is empty in the response, all data within the range is returned.

Examples

-
-
-

3.6. Auto-increment of primary key columns

This topic describes how to use an auto-increment primary key column. You cannot set a partition key to an auto-increment column. If you write data to a table that contains an auto-increment primary key column, you do not need to specify a specific value for the auto-increment primary key column because Tablestore generates a value for the auto-increment primary key column. The value generated for the auto-increment primary key column is unique, and all values in auto-increment primary key columns increase sequentially within a partition that shares the same partition key value.

Features

The auto-increment function of the primary key column has the following features:

- The values of auto-increment primary key columns are unique and increase sequentially but not always continuously within a partition that shares the same partition key value.
- The value of an auto-increment primary key column is 64-bit signed long integer.
- You can create an auto-increment primary key column for a table. An instance can include tables that contain and tables that do not contain auto-increment primary key columns.

> **Note** The auto-increment function of the primary key column does not affect the rule of conditional update. For more information about conditional update, see Conditional update.

Limits

The auto-increment function of the primary key column has the following limits:

- You can create at most one auto-increment primary key column for a table. You cannot set a partition key to an auto-increment column.
- You can create an auto-increment primary key column only when you create a table. You cannot create an auto-increment primary key column for an existing table.
- An auto-increment primary key column can only be an integer column. The generated value for an auto-increment primary key column is 64-bit signed long integer.
- You can not set an attribute column to an auto-increment primary key column.

API operations

The following table describes the API operations for the auto-increment function of the primary key column.

API operation	Description

API operation	Description
CreateTable	When you create a table, you cannot set the partition key to be an auto-increment primary key column. If you set the partition key to be an auto-increment primary key column, values in the column cannot be automatically generated.
UpdateTable	After a table is created, you cannot use the UpdateTable operation to change a primary key column to an auto-increment primary key column.
PutRow	When you write data to a table, you do not need to specify a specific value for the auto-increment primary key column. Tablestore generates a value for the auto-increment primary key column. You can set ReturnType to RT_PK to obtain values of all primary key columns and use the values in GetRow to query data.
UpdateRow	
BatchWriteRow	
GetRow	You must use values of all primary key columns when you use GetRow. To obtain values of all primary key columns, you can set ReturnType to RT_PK in PutRow, UpdateRow, or BatchWriteRow.
BatchGetRow	

Examples

The use of the auto-increment function of the primary key column involves the CreateTable, PutRow, UpdateRow, and BatchWriteRow operations.

1. Create a table

   To create an auto-increment primary key column when you create a table, you must set the attribute of the primary key column to AUTO_INCREMENT.

```
private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(“table_name”);
        // Create the first primary key column, which is also the partition key.
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_1", PrimaryKeyType.STRIN
G));
        // Create the second primary key column, and set it to an auto-increment column
. The type is set to INTEGER, and the attribute is set to AUTO_INCREMENT.
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_2", PrimaryKeyType.INTEG
ER, PrimaryKeyOption.AUTO_INCREMENT));
        int timeToLive = -1;  // Specify that data never expires.
        int maxVersions = 1;  // Specify that only one version of data is saved.
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
        client.createTable(request);
    }
```

2. Write data

   When you write data to a table, you do not need to specify a specific value for the auto-increment primary key column. Instead, you need only to set the value in the auto-increment primary key column to AUTO_INCREMENT.

```
    private static void putRow(SyncClient client, String receive_id) {
        // Create the primary key.
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder
();
        // Set the value in the first primary key column to the first four digits of md
5(receive_id).
        primaryKeyBuilder.addPrimaryKeyColumn("PK_1", PrimaryKeyValue.fromString("Hangz
hou"));
        // Set the value in the second primary key column to AUTO_INCREMENT. The second
primary key column is an auto-increment primary key column, and you do not need to spec
ify a specific value for it. Tablestore generates a value for the auto-increment primar
y key column.
        primaryKeyBuilder.addPrimaryKeyColumn("PK_2", PrimaryKeyValue.AUTO_INCREMENT);
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        RowPutChange rowPutChange = new RowPutChange("table_name", primaryKey);
        // Set ReturnType to RT_PK to include the primary key column values in the retu
rned result. By default, no primary key column values are returned if ReturnType is not
set.
        rowPutChange.setReturnType(ReturnType.RT_PK);
        // Add attribute columns.
        rowPutChange.addColumn(new Column("content", ColumnValue.fromString(content)));
        // Write data to the table.
        PutRowResponse response = client.putRow(new PutRowRequest(rowPutChange));
        // Display the returned primary key value.
        Row returnRow = response.getRow();
        if (returnRow != null) {
            System.out.println("PrimaryKey:" + returnRow.getPrimaryKey().toString());
        }
        // Display the consumed capacity units (CUs).
        CapacityUnit  cu = response.getConsumedCapacity().getCapacityUnit();
        System.out.println("Read CapacityUnit:" + cu.getReadCapacityUnit());
        System.out.println("Write CapacityUnit:" + cu.getWriteCapacityUnit());
    }
```

Usage

You can use the following Tablestore SDKs to implement the auto-increment function of the primary key column:

- Tablestore SDK for Java: Configure an auto-increment primary key column
- Tablestore SDK for Go: Configure an auto-increment primary key column
- Tablestore SDK for Python: Configure an auto-increment primary key column
- Tablestore SDK for Node.js: Configure an auto-increment primary key column
- Tablestore SDK for .NET: Configure an auto-increment primary key column
- Tablestore SDK for PHP: Configure an auto-increment primary key column

Billing methods

The implementation of the auto-increment function of the primary key column does not affect the current billing rules. Values of returned primary key columns do not consume additional read CUs.

3.7. Conditional update

If you use conditional update, data in the table can be updated only when the conditions are met. If the conditions are not met, the update fails.

Scenarios

Conditional update applies to scenarios where high-concurrency applications are updated.

In these scenarios, old_value may be updated by other clients. If you use conditional update, the current value is updated to new_value only when the current value is equal to old_value.

> ? **Note** In scenarios where concurrency is high, such as web page counting or games, data updates may fail if you use conditional update. In these cases, you can retry the data update.

```
// Obtain the current value.
old_value = Read();
// Perform calculations on the current value, such as add 1 to the current value.
new_value = func(old_value);
// Use the new value to update the current value.
Update(new_value);
```

Operations

Conditional update supports relational operator-based operations including =, !, =, >, >=, <, and <= and logical operations including NOT, AND, and OR. You can use a combination of up to 10 conditions in an update operation. You can use conditional update in the PutRow, UpdateRow, DeleteRow, and BatchWriteRow operations.

Conditional update can be used to implement optimistic locking. When you update a row, the value of the specified column is obtained. For example, Column A has a value of 1. Obtain the value in Column A and set a condition that **the value of Column A is 1**. Update **the value of Column A to 2**. If the row is updated by another client, the update fails.

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

Column-based judgment condition	Description
Column-based condition	Column-based conditions support SingleColumnValueCondition and CompositeColumnValueCondition, which are used to perform the condition-based judgment based on the values of one or more columns. Column-based conditions are similar to the conditions used by Tablestore filters.
Row existence condition	When you modify a table, Tablestore first checks the row existence condition. If the row existence condition is not met, the modification fails and an error is reported. The row existence condition is classified into the following types: • IGNORE • EXPECT_EXIST • EXPECT_NOT_EXIST For more information, see Row existence condition-based update rules.

Row existence condition-based update rules

Note BatchWriteRow is a set of multiple PutRow, UpdateRow, and DeleteRow operations. When you manage data in the table by using the BatchWriteRow operation, see the update rules of the corresponding operation based on the operation type.

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

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement conditional update:

- Tablestore SDK for Java: Conditional update
- Tablestore SDK for Go: Conditional update
- Tablestore SDK for Python: Conditional update
- Tablestore SDK for Node.js: Conditional update
- Tablestore SDK for .NET: Conditional update
- Tablestore SDK for PHP: Conditional update

Examples

The following code provides examples on how to use column-based judgment conditions and implement optimistic locking:

- Construct a SinglleColumnValueCondition.

```
 // Set the condition: Col0==0.
 SingleColumnValueCondition singleColumnValueCondition = new SingleColumnValueCondition("
Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(0));
 // If Col0 does not exist, the condition is not met.
 singleColumnValueCondition.setPassIfMissing(false);
 // Specify that only the latest version is used for comparison.
 singleColumnValueCondition.setLatestVersionsOnly(true);
```

- Construct a CompositeColumnValueCondition.

```
 // Set condition composite1 to (Col0 == 0) AND (Col1 > 100).
 CompositeColumnValueCondition composite1 = new CompositeColumnValueCondition(CompositeCo
lumnValueCondition.LogicOperator.AND);
 SingleColumnValueCondition single1 = new SingleColumnValueCondition("Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(0));
 SingleColumnValueCondition single2 = new SingleColumnValueCondition("Col1",
         SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong(10
0));
 composite1.addCondition(single1);
 composite1.addCondition(single2);
 // Set condition composite2 to ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10).
 CompositeColumnValueCondition composite2 = new CompositeColumnValueCondition(CompositeCo
lumnValueCondition.LogicOperator.OR);
 SingleColumnValueCondition single3 = new SingleColumnValueCondition("Col2",
         SingleColumnValueCondition.CompareOperator.LESS_EQUAL, ColumnValue.fromLong(10))
;
 composite2.addCondition(composite1);
 composite2.addCondition(single3);
```

- The following code provides an example on how to implement optimistic locking by increasing the value of a column:

```
 private static void updateRowWithCondition(SyncClient client, String pkValue) {
     // Construct the primary key.
     PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
     primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(p
kValue));
     PrimaryKey primaryKey = primaryKeyBuilder.build();
     // Read a row of data.
     SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey)
;
     criteria.setMaxVersions(1);
     GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
     Row row = getRowResponse.getRow();
     long col0Value = row.getLatestColumn("Col0").getValue().asLong();
     // Configure conditional update to increase the value of Col0 by 1.
     RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
     Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
     ColumnCondition columnCondition = new SingleColumnValueCondition("Col0", SingleColum
nValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(col0Value));
     condition.setColumnCondition(columnCondition);
     rowUpdateChange.setCondition(condition);
     rowUpdateChange.put(new Column("Col0", ColumnValue.fromLong(col0Value + 1)));
     try {
         client.updateRow(new UpdateRowRequest(rowUpdateChange));
     } catch (TableStoreException ex) {
         System.out.println(ex.toString());
     }
 }
```

Billing methods

The calculation of capacity units (CUs) is not affected if the data is written or updated. However, if the conditional update fails, a write CU and a read CU are consumed.

3.8. Local transactions

This topic describes how to use the local transaction feature. You can create a local transaction based on a specified partition key value. After you read or write data within a local transaction, you can commit or abort the local transaction. Pessimistic locking is used to control concurrent operations within a local transaction.

The local transaction feature is available for invitational preview. By default, this feature is disabled. To use the local transaction feature, submit a ticket to apply for invitational preview.

The local transaction feature allows you to perform atomic operations to read or write one or more rows.

Scenarios

- Read and write operations (simple scenarios)

  You can use the following methods to perform read, modify, and write (RMW) operations. Different methods have different limits.

  - Conditional update: processes only one request that involves a single row at a time. It cannot be used to process requests that involve data across rows or requests for multiple write operations. For more information, see Conditional update.
  - Atomic counter: processes only one request that involves a single row at a time, and supports only the increment of column values. For more information, see Atomic counters.

  You can create a local transaction to perform an RMW operation on data within the range specified based on a partition key value.

  i. Use StartLocalTransaction to create a local transaction based on a specified partition key value and obtain the ID of the local transaction.

  ii. Use GetRow or GetRange to read data. The transaction ID must be included in the request.

  iii. Modify data on the client.

  iv. Use PutRow, UpdateRow, DeleteRow, or BatchWriteRow to write back the modified data. The transaction ID must be included in the request.

  v. Use CommitTransaction to commit the transaction.

- Email (complex scenarios)

  You can create a local transaction to perform atomic operations on emails of the same user.

  To use the local transaction feature, include a data table and two index tables in a physical table. The following table describes the primary key columns of the data table and index tables.

  The Type column is used to identify the rows of the data table and the rows of index tables. The IndexField column stores the $Folder field for the Folder index table, and the $SendTime field for the SendTime index table. The IndexField column is empty for the data table.

Table	UserID	Type	IndexField	MailID
Data table	User ID	"Main"	"N/A"	Email ID
Folder table	User ID	"Folder"	$Folder	Email ID
SendTime table	User ID	"SendTime"	$SendTime	Email ID

You can use a local transaction to perform the following operations:

- Perform the following steps to list the last 100 emails sent by a user:
  a. Use the user ID to create a local transaction and obtain the transaction ID.
  b. Include the transaction ID in the request when you call GetRange to query 100 emails from the SendTime table.
  c. Include the transaction ID in the request when you call BatchGetRow to query the detailed information of the 100 emails from the data table.
  d. Use CommitTransaction to commit the local transaction or use AbortTransaction to abort the local transaction.

     The commit and abort operations have the same effect because no writes are performed in this transaction.
- Perform the following steps to transfer all emails from a folder to another folder:
  a. Use the user ID to create a local transaction and obtain the transaction ID.
  b. Include the transaction ID in the request when you call GetRange to query multiple emails from the Folder table.
  c. Include the transaction ID in the request when you call BatchWriteRow to perform write operations on the Folder table.

     The write operation is performed on two rows each time when an email is transferred. Specifically, a row that indicates the original folder is deleted from the Folder table and a row that indicates the new folder is added to the Folder table.
  d. Use CommitTransaction to commit the transaction.
- Perform the following steps to count the numbers of read emails and unread emails in a folder. This method is not the most efficient solution.
  a. Use the user ID to create a local transaction and obtain the transaction ID.
  b. Include the transaction ID in the request when you call GetRange to query multiple emails from the Folder table.
  c. Include the transaction ID in the request when you call BatchGetRow to query the read status of each email from the data table.
  d. Use CommitTransaction to commit the local transaction or use AbortTransaction to abort the local transaction.

     The commit and abort operations have the same effect because no write operations are performed in this transaction.

In this scenario, you can add more index tables to accelerate common operations. The local transaction feature ensures the consistency between the status of the data table and index tables, which simplifies development. For example, when you count the number of emails, many emails are read, which results in high overheads. To reduce overheads and accelerate queries, you can use a new index table to store the numbers of read emails and unread emails.

Limits

- The validity period of a local transaction is up to 60 seconds.

  If a transaction is not committed or aborted within 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.

- A transaction may be created on the Tablestore server even if a timeout error is returned. In this case, you can resend a transaction creation request after the created transaction times out.
- If a local transaction is not committed, it may become invalid. In this case, retry the operation within this transaction.
- Tablestore imposes the following limits on read and write operations on data within a local transaction:
  - The transaction ID cannot be used to access data beyond the range specified based on the partition key value that is used to create the transaction.
  - The partition key values of all write requests in the same transaction must be the same as the partition key value used to create the transaction. This limit does not apply to read requests.
  - A local transaction can be used only by one request at a time. When the transaction is in use, other operations that use the transaction ID fail.
  - The maximum interval for read and write operations on data within a transaction is 60 seconds.

    If a transaction is not read or written for more than 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.

  - Up to 4 MB of data can be written to each transaction. The volume of data written to each transaction is calculated in the same way as a regular write request.
  - If you do not specify a version number for a cell, the Tablestore server assigns a version number to the cell in the usual way when the cell is written to the transaction (rather than when the transaction is committed).
  - If a BatchWriteRow request includes a transaction ID, all rows in the request can be written only to the table that matches the transaction ID.
  - When you use a transaction, the data within the range specified based on the corresponding partition key value is locked. If a request that is sent to write data within the local transaction does not contain the transaction ID, the request fails. Data within the transaction is unlocked when the transaction is committed or aborted, or when the transaction times out.
  - A transaction remains valid even if a read or write request with the transaction ID is rejected. You can resend the request in the same manner as a regular request or you can abort the transaction.

Operations

The following table describes the operations that can be performed on local transactions.

Operation	Description
StartLocalTransaction	Creates a local transaction.

Operation	Description
CommitTransaction	Commits a local transaction.
AbortTransaction	Aborts a local transaction.
GetRow	Read and write data within a local transaction. For more information, see Single-row operations and Multi-row operations. Note Data within a local transaction is within a range specified based on a partition key value. For more information about partition keys, see Primary keys and attributes.
PutRow	
UpdateRow	
DeleteRow	
BatchWriteRow	
GetRange	

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement local transactions:

- Tablestore SDK for Java: Local transactions
- Tablestore SDK for Go: Local transactions
- Tablestore SDK for Python: Local transactions
- Tablestore SDK for Node.js: Local transactions
- Tablestore SDK for PHP: Local transactions

Parameters

Parameter	Description
TableName	The name of the data table.
PrimaryKey	The primary key of the data table. • You must specify a partition key value when you create a local transaction. • You must specify all primary key columns when you read and write data in a local transaction.
TransactionId	The local transaction ID that identifies the local transaction. You must specify a transaction ID when you read and write data within the local transaction.

Examples

1. Call the startLocalTransaction method of AsyncClient or SyncClient to create a local transaction based on a specified partition key value and obtain the ID of the created local transaction.

```
PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
StartLocalTransactionRequest request = new StartLocalTransactionRequest(tableName, prim
aryKey);
String txnId = client.startLocalTransaction(request).getTransactionID();
```

2. Read and write data within a local transaction.

   You must specify the transaction ID to read and write data within the local transaction. The read and write operations are similar to regular operations.

   - Write a row to the local transaction.

```
PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
primaryKeyBuilder.addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong("userId"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
RowPutChange rowPutChange = new RowPutChange(tableName, primaryKey);
rowPutChange.addColumn(new Column("Col", ColumnValue.fromLong(columnValue)));
PutRowRequest request = new PutRowRequest(rowPutChange);
request.setTransactionId(txnId);
client.putRow(request);
```

   - Read the row from the local transaction.

```
PrimaryKeyBuilder primaryKeyBuilder;
primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
primaryKeyBuilder.addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong("userId"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(tableName, primaryKey);
criteria.setMaxVersions(1); // Specify that the latest version of data is read.
GetRowRequest  request = new GetRowRequest(criteria);
request.setTransactionId(txnId);
GetRowResponse getRowResponse = client.getRow(request);
```

3. Commit or abort a local transaction.

   - Commit a local transaction so that all data modifications within the local transaction take effect.

```
CommitTransactionRequest commitRequest = new CommitTransactionRequest(txnId);
client.commitTransaction(commitRequest);
```

   - Abort a local transaction so that all data modifications within the local transaction do not take effect.

```
AbortTransactionRequest abortRequest = new AbortTransactionRequest(txnId);
client.abortTransaction(abortRequest);
```

Billing

- Each StartLocalTransaction, CommitTransaction, and AbortTransaction operation consumes one write capacity unit (CU).
- Requests sent to read or write data in local transactions are billed in the same way as regular read and write requests. For more information, see Billing overview.

Error codes

Error code	Description
OTSRowOperationConflict	The error message returned because the specified partition key value is used by an existing local transaction.
OTSSessionNotExist	The error message returned because the transaction that has the specified transaction ID does not exist, or the specified transaction is invalid or timed out.
OTSSessionBusy	The error message returned because the last request on the transaction is incomplete.
OTSOutOfTransactionDataSizeLimit	The error message returned because the amount of data within a transaction exceeds the maximum volume.
OTSDataOutOfRange	The error message returned because data operation is beyond the range specified by the partition key value used to create the transaction.

3.9. Atomic counters

Atomic counters allow you to implement an atomic counter on a column. This feature provides statistic data for online applications such as the number of page views (PVs) on various topics.

Atomic counters reduce the write performance overhead caused by forced consistency. When you send a request to the server to perform read, modify, and write (RMW) operations, the server 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.

Scenarios

You can use an atomic counter to keep track of a row in real time.

Assume that you create a table to store metadata of pictures. Each row in the table has a user ID. A column of the row is used to store metadata of pictures. Another column of the row is used as an atomic counter to count the number of pictures whose metadata is stored in the row.

- When you use UpdateRow to add metadata of a picture to a row, the atomic counter is increased by one.
- When you use UpdateRow to delete metadata of a picture from a row, the atomic counter is decreased by one.
- You can use GetRow to read the value of the atomic counter to check the number of pictures whose metadata is stored in the row.

This function ensures database consistency. When you add metadata of a picture to a row, the atomic counter value of the row is increased by one instead of decreased by one.

Note An error may occur when an atomic counter encounters network timeouts or system failures. In this case, you can retry the operation. However, the atomic counter may be updated twice, which results in a smaller or greater value of the atomic counter. In this case, we recommend that you use Conditional update to precisely update the value of a column.

Limits

- You can implement atomic counters only on INTEGER columns.
- By default, if a column that is specified as an atomic counter does not exist, the value of the column is 0 before you write data. If a column that is specified as an atomic counter is not an INTEGER column, an OTSParameterInvalid error occurs.
- You can update an atomic counter by using a positive or negative number, but you must avoid an integer overflow. If an integer overflow occurs, an OTSParameterInvalid error is returned.
- By default, the value of an atomic counter is not returned in the response to an update row request. You can specify that the increased value of an atomic counter is returned.
- You cannot specify a column as an atomic counter and update the column in a single request. If Attribute Column A is set to an atomic counter, you cannot perform other operations such as overwrite and delete operations on the attribute column A.
- You can perform multiple update operations on the same row by using a BatchWriteRow request. However, if you perform an atomic counter operation on a row, you can perform only one update operation on the row in a BatchWriteRow request.
- Only the value of the latest version of an atomic counter is increased. You cannot increase the value of a specified version of an atomic counter. After you update a row, a new version of data is inserted to the atomic counter in the row.

API operations

The following table describes the operations added to the rowUpdateChange class to perform atomic counters.

API operation	Description
RowUpdateChange increment(Column column)	Increments or decreases the value in a column by a number.
void addReturnColumn(String columnName)	Specifies the name of the atomic counter to return its value.
void setReturnType(ReturnType returnType)	Specifies a data type to return the value of an atomic counter.

Usage

You can use the following Tablestore SDKs to implement atomic counters:

- Tablestore SDK for Java: Atomic counters
- Tablestore SDK for Go: Atomic counters
- Tablestore SDK for Python: Atomic counters
- Tablestore SDK for Node.js: Atomic counters
- Tablestore SDK for .NET SDK: Atomic counters

Parameters

Parameter	Description
tableName	The name of the table.

Parameter	Description
columnName	The name of the column you set to an atomic counter. You can specify only INTEGER columns as atomic counters.
value	The value you increase to or decrease from the atomic counter value.
returnType	If you set this parameter to ReturnType.RT_AFTER_MODIFY, the value of the atomic counter is returned.

Examples

The following code provides an example on how to use rowUpdateChange to increase the value of an atomic counter and return the increased value:

```
private static void incrementByUpdateRowApi(SyncClient client) {
        // Specify the primary key.
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
        primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(
"pk0"));
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        // Specify the table.
        RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
        // Set the price column as an atomic counter and increase the value of the atomic c
ounter by 10. You cannot specify the timestamp.
        rowUpdateChange.increment(new Column("price", ColumnValue.fromLong(10)));
        // Set the data type of the value to return to ReturnType.RT_AFTER_MODIFY and retur
n the value of the atomic counter.
        rowUpdateChange.addReturnColumn("price");
        rowUpdateChange.setReturnType(ReturnType.RT_AFTER_MODIFY);
        // Initiate a request to update the row.
        UpdateRowResponse response = client.updateRow(new UpdateRowRequest(rowUpdateChange)
);
        // Display the updated values.
        Row row = response.getRow();
        System.out.println(row);
    }
```

Billing methods

The implementation of atomic counters does not affect the existing billing methods.

3.10. Configure a filter

Tablestore filters results on the server before returning the rows that match the filter conditions. This feature reduces the volume of data transferred and shortens the response time because only matched rows are returned.

Scenarios

- Directly filter results

An Internet of things (IoT)-based smart electric meter writes voltage, current, usage, and other information to a Tablestore table every 15 seconds. You want to query abnormal voltage data and other related data for daily analysis to determine whether to inspect cables.

You can use GetRange to read the monitoring data generated by the electric meter and filter the data (5,760 records) to obtain the 10 records that are collected when the voltage is unstable.

If you use a filter, only the 10 records that need to be analyzed are returned. A filter reduces the volume of returned data and removes the need for preliminary data processing. This reduces development costs.

- Filter results after regular expression matching and data conversion

  When data is stored in a custom format such as JSON string and you want to query a subfield value, you can use regular expressions to match and then convert the value into the data type you require. Then, you can use a filter to obtain the required data.

  For example, the data stored in a column is in the format of `{cluster_name:name 1,lastupdatetime:12345}`. If you need to filter and query the value of row lastupdatetime>12345, you can use the regular expression `lastupdatetime:([0-9]+)}` to match the data of the subfield in the column, use the CAST function to convert the matching results into a numeric type, and then compare the numeric data to the matched results. This way, you can obtain the required data row.

Limits

- Filters support relational operator-based operations (=, !=, >, >=, <, and <=) and logical operations (NOT, AND, and OR). You can use a combination of up to 10 filter conditions for a filter.
- The reference columns that are used by a filter must be included in the read data. If the specified columns from which data is read do not include reference columns, the filter cannot query the values of reference columns.
- When you use the GetRange operation, up to 5,000 rows or 4 MB of data can be scanned at a time.

  If no data matches the filter conditions in the range of the scan, the returned rows are empty. However, NextStartPrimaryKey may not be empty. If NextStartPrimaryKey is not empty, use this value to continue scanning until the return value of NextStartPrimaryKey is empty.

Operations

Filters can be used for the GetRow, BatchGetRow, and GetRange operations. You can use filters by calling the GetRow, BatchGetRow, and GetRange operations, which does not change the native semantics or limits of these operations. For more information, see Single-row operations and Multi-row operations.

The available filters are SingleColumnValueFilter, SingleColumnValueRegexFilter, and CompositeColumnValueFilter, which filter a row based on the column values of one or more reference columns.

- SingleColumnValueFilter determines whether to filter a row based on the value of a reference column.

  SingleColumnValueFilter uses the PassIfMissing parameter to determine whether the filter conditions are met if the reference column does not exist. You can specify an action when the reference column does not exist.

- SingleColumnValueRegexFilter uses regular expressions to match column values of the String type and extract matching substrings. Then, this filter converts the data type of the extracted substrings to String, Integer, or Double and filters the values after conversion.

Notice Only Tablestore SDK for Java supports the SingleColumnValueRegexFilter filter.

- CompositeColumnValueFilter determines whether to filter a row based on a logical combination of the check results for the values of multiple reference columns.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement filters:

- Tablestore SDK for Java: Filter
- Tablestore SDK for Go: Filter
- Tablestore SDK for Python: Filter
- Tablestore SDK for Node.js: Filter
- Tablestore SDK for .NET: Filter
- Tablestore SDK for PHP: Filter

Parameters

Parameter	Description
ColumnName	The name of the reference column used by a filter.
ColumnValue	The value of the reference column used by a filter.
CompareOperator	The relational operator used by a filter. Relational operators include EQUAL (=), NOT_EQUAL (!=),GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=).
LogicOperator	The logical operator used by a filter. Logical operators include NOT, AND, and OR.
PassIfMissing	Specifies whether to return a row when a reference column does not exist in the row. Valid values: • true: If the reference column does not exist in a row, the row is returned. This is the default value. • false: If the reference column does not exist in a row, the row is not returned.
LatestVersionsOnly	Specifies whether to use only the latest versions of data in a reference column for comparison when the reference column contains data of multiple versions. The value of this parameter is of the Bool type. The default value is true. If the default value is used, the latest versions of data are used for comparison when a reference column contains data of multiple versions. If the value of LatestVersionsOnly is set to false, all versions of data in a reference column are used for comparison. If one version of data in the reference column meets the filter conditions, the row is returned.

Parameter	Description
Regex	A regular expression used to match subfield values. The regular expression must meet the following conditions: • A regular expression can be up to 256 bytes in length. • The syntax of regular expressions in Perl is supported. • Single-byte regular expressions are supported. • Regular expression matching in Chinese is not supported. • Full matching mode and partial matching mode of regular expressions are supported. In partial matching mode, regular expressions are separated by a pair of parentheses (). If the full matching mode is used, the first matching result is returned. If particle matching mode is used, the first submatch is returned. For example, if the column value is 1aaa51bbb5 and the regular expression is 1[a-z]+5, the return value is 1aaa5. If the regular expression is 1([a-z]+)5, the return value is aaa.
VariantType	The data type of the subfield value after conversion when you use a regular expression to match the subfield value. Valid values: VT_INTEGER (integer), VT_STRING (string type), and VT_DOUBLE (double-precision floating-point type).

Examples

- Construct SingleColumnValueFilter.

```
 // // Configure a filter to return a row when the value of Col0 is 0
 SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
         SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
 // If Col0 does not exist, the row is not returned.
 singleColumnValueFilter.setPassIfMissing(false);
 // Only the latest version of data in the column is used for comparison.
 singleColumnValueFilter.setLatestVersionsOnly(true);
```

- Construct SingleColumnValueRegexFilter.

```
 // Construct a rule to extract regular expressions.
 RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
 // Set a filter to implement cast<int>(regex(col1) >0.
 // The SingleColumnValueRegexFilter constructing is in the format of column name, regula
r rule, comparison character, comparison value.
 SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
     regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(0));
 // If Col0 does not exist, the row is not returned.
 filter.setPassIfMissing(false);
```

- Construct CompositeColumnValueFilter.

```
  // Set the composite1 condition to (Col0 == 0) AND (Col1 > 100).
  CompositeColumnValueFilter composite1 = new CompositeColumnValueFilter(CompositeColumnV
alueFilter.LogicOperator.AND);
  SingleColumnValueFilter single1 = new SingleColumnValueFilter("Col0",
          SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
  SingleColumnValueFilter single2 = new SingleColumnValueFilter("Col1",
          SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromLong(100)
);
  composite1.addFilter(single1);
  composite1.addFilter(single2);
  // Set the composite2 condition to ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10).
  CompositeColumnValueFilter composite2 = new CompositeColumnValueFilter(CompositeColumnV
alueFilter.LogicOperator.OR);
  SingleColumnValueFilter single3 = new SingleColumnValueFilter("Col2",
          SingleColumnValueFilter.CompareOperator.LESS_EQUAL, ColumnValue.fromLong(10));
  composite2.addFilter(composite1);
  composite2.addFilter(single3);
```

Billing

The implementation of filters does not affect existing billing rules.

Although filters reduce the volume of returned data, the disk I/O usage remain unchanged because filtering is performed on the server side before data is returned. Therefore, the same number of read CUs are consumed regardless of whether the filters are used or not. For example, when you use GetRange to read 100 records (200 KB) of data and then filter these records to obtain 10 records (20 KB), 50 read CUs are consumed.

4.Timeline model

4.1. Overview

The Timeline model is designed for message data. This model can meet the specific requirements of message data, such as message order preservation, storage of large numbers of messages, and real-time synchronization. The Timeline model also supports tokenization and BoolQuery. The model is suitable for 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 the Timeline model to implement your services in different methods as required. The architecture of the model includes the following components:

- Store: a unit used to store Timeline data. A store is similar to a table in database services.
- 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 the messages in a Timeline. A Timeline can include one or multiple queues.
- 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 value manually.
- 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 BoolQuery.

Functions

The Timeline model supports the following features:

- Management of Meta data and messages, including basic data operations such as create, read, update, and delete.
- BoolQuery and tokenization for Meta data and messages.
- Two configuration methods for SequenceId values: auto-increment column and manual setting.
- Timeline Identifier that contains multiple columns.
- Compatibility with the Timeline model V1.x. The TimelineMessageForV1 examples of the Timeline model can be used to read messages from and write messages to the Timeline model V1.x.

If you use Tablestore SDK for Java 4.12.1 or later (in which the Timeline model is integrated), add the following dependency to use the Timeline model:

```
<dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>tablestore</artifactId>
    <version>4.12.1</version>
</dependency>
```

If you use Tablestore SDK for Java earlier than 4.12.1, add the following dependency to use the Timeline model:

```
<dependency>
    <groupId>com.aliyun.openservices.tablestore</groupId>
    <artifactId>Timeline</artifactId>
    <version>2.0.0</version>
</dependency>&nbsp;
```

4.2. Quick start

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

Procedure

1. Log on to the Tablestore console. Create a Tablestore instance. For more information, see Create instances.
2. Download and install Tablestore SDK for Java. For more information, see Install Tablestore SDK for Java.
3. Use an endpoint and configure an AccessKey pair to initialize an OTSClient instance. For more information, see Initialization.
4. Download the sample code to get started with the Timeline model.

4.3. Basic operations

4.3.1. Overview

The Timeline model is a data model designed for messaging applications. This model has many specialized features such as message order preservation, storage of large numbers of messages, and real-time synchronization to effectively implement messaging functions. The model also supports full-text search and bool query. The model is also suitable for instant messaging (IM) and feed stream scenarios. The Timeline model Java SDK includes the following operations:

- Initialization
- Meta management
- Timeline management
- Queue management

4.3.2. Initialization

This topic describes how to initialize the Timeline model.

Initialize TimelineStoreFactory

Use SyncClient as a parameter to initialize TimelineStoreFactory and create a Store to manage Meta data and Timeline data. The retry operation required 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 call the RetryStrategy operation to customize the retry policy.

```
/**
 * Configure the retry policy.
 * Code: configuration.setRetryStrategy(new DefaultRetryStrategy());
 * */
ClientConfiguration configuration = new ClientConfiguration();
SyncClient client = new SyncClient(
        "http://instanceName.cn-shanghai.ots.aliyuncs.com",
        "accessKeyId",
        "accessKeySecret",
        "instanceName", configuration);
TimelineStoreFactory serviceFactory = new TimelineStoreFactoryImpl(client);
```

Initialize TimelineMetaStore

Create a schema for a Meta table. The schema includes parameters such as Identifier and MetaIndex. Create and obtain a Store to manage Meta data by using TimelineStoreFactory. You must specify the following parameters: Meta table name, index name, primary key field, and index type.

```
TimelineIdentifierSchema idSchema = new TimelineIdentifierSchema.Builder()
        .addStringField("timeline_id").build();
IndexSchema metaIndex = new IndexSchema();
metaIndex.addFieldSchema( //Set the index field and index type.
        new FieldSchema("group_name", FieldType.TEXT).setIndex(true).setAnalyzer(FieldSchem
a.Analyzer.MaxWord),
        new FieldSchema("create_time", FieldType.Long).setIndex(true)
);
TimelineMetaSchema metaSchema = new TimelineMetaSchema("groupMeta", idSchema)
        .withIndex("metaIndex", metaIndex); //Set the index.
TimelineMetaStore timelineMetaStore = serviceFactory.createMetaStore(metaSchema);
```

- Create a table

  Create a table based on the parameters in metaSchema. If an index is configured in metaSchema, the index is created after the table is created.

  ```
  timelineMetaStore.prepareTables();
  ```

- Delete a table

  If a table contains an index, the index is deleted before the table.

  ```
  timelineMetaStore.dropAllTables();
  ```

Initialize TimelineStore

Create a schema for a Timeline table. The schema includes parameters such as Identifier and TimelineIndex. Create and obtain a Store to manage Timeline data by using TimelineStoreFactory. You must specify the following parameters: Timeline table name, index name, primary key field, and index type.

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

```
TimelineIdentifierSchema idSchema = new TimelineIdentifierSchema.Builder()
        .addStringField("timeline_id").build();
IndexSchema timelineIndex = new IndexSchema();
timelineIndex.setFieldSchemas(Arrays.asList(//Configure the index field and index type.
        new FieldSchema("text", FieldType.TEXT).setIndex(true).setAnalyzer(FieldSchema.Anal
yzer.MaxWord),
        new FieldSchema("receivers", FieldType.KEYWORD).setIndex(true).setIsArray(true)
));
TimelineSchema timelineSchema = new TimelineSchema("timeline", idSchema)
        .autoGenerateSeqId() //Specify the auto-increment column as the method to generate
the SequenceId value.
        .setCallbackExecuteThreads(5) //Set the number of initial threads of DefaultTableSt
oreWriter to 5.
        .withIndex("metaIndex", timelineIndex); //Set the index.
TimelineStore timelineStore = serviceFactory.createTimelineStore(timelineSchema);
```

- Create a table

  Create a table based on the parameters in TimelineSchema. If an index is configured in TimelineSchema, the index is created after the table is created.

  ```
  timelineStore.prepareTables();
  ```

- Delete a table

  If a table contains an index, the index is deleted before the table.

  ```
  timelineStore.dropAllTables();
  ```

4.3.3. Meta management

You can call 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 of the 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 Overview.

Insert

The TimelineIdentifier value is used to uniquely identify Timeline data. Tablestore overwrites repeated TimelineIdentifier values.

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
TimelineMeta meta = new TimelineMeta(identifier)
        .setField("filedName", "fieldValue");
timelineMetaStore.insert(meta);
```

Read

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

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
timelineMetaStore.read(identifier);
```

Update

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

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
TimelineMeta meta = new TimelineMeta(identifier)
        .setField("filedName", "new value");
timelineMetaStore.update(meta);
```

Delete

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

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
timelineMetaStore.delete(identifier);
```

Search

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

```
/**
 * Search meta by SearchParameter.
 * */
SearchParameter parameter = new SearchParameter(
        field("fieldName").equals("fieldValue")
);
timelineMetaStore.search(parameter);
/**
 * Search meta by SearchQuery.
 * */
TermQuery query = new TermQuery();
query.setFieldName("fieldName");
query.setTerm(ColumnValue.fromString("fieldValue"));
SearchQuery searchQuery = new SearchQuery().setQuery(query);
timelineMetaStore.search(searchQuery);
```

4.3.4. Timeline management

You can call the operations for fuzzy query and Boolean 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 of the 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 Overview.

Search

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

```
/**
 * Search timeline by SearchParameter.
 * */
SearchParameter searchParameter = new SearchParameter(
        field("text").equals("fieldValue")
);
timelineStore.search(searchParameter);
/**
 * Search timeline by SearchQuery.
 * */
TermQuery query = new TermQuery();
query.setFieldName("text");
query.setTerm(ColumnValue.fromString("fieldValue"));
SearchQuery searchQuery = new SearchQuery().setQuery(query).setLimit(10);
timelineStore.search(searchQuery);
```

Flush

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

```
/**
 * Flush messages in buffer, and wait until all messages are stored.
 * */
timelineStore.flush();
```

4.3.5. Queue management

This topic describes how to manage queues when you use the Timeline model.

Obtain a Queue instance

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

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group_1")
        .build();
// The Queue corresponds to an identifier of a TimelineStore.
TimelineQueue timelineQueue = timelineStore.createTimelineQueue(identifier);
```

The Queue instance manages a message queue that corresponds to an identifier of a TimelineStore. This instance provides 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 manually or by using an auto-increment column.

```
timelineQueue.store(message);//Generate the SequenceId value by using an auto-increment column.
timelineQueue.store(sequenceId, message);//Manually set the SequenceId value.
```

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>.

```
TimelineCallback callback = new TimelineCallback() {
    @Override
    public void onCompleted(TimelineIdentifier i, TimelineMessage m, TimelineEntry t) {
        // do something when succeed.
    }
    @Override
    public void onFailed(TimelineIdentifier i, TimelineMessage m, Exception e) {
        // do something when failed.
    }
};
timelineQueue.storeAsync(message, callback);//Generate the SequenceId value by using an auto-increment column.
timelineQueue.storeAsync(sequenceId, message, callback);//Manually set the SequenceId value.
```

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.

```
timelineQueue.batchStore(message);//Generate the SequenceId value by using an auto-increment column.
timelineQueue.batchStore(sequenceId, message);//Manually set the SequenceId value.
timelineQueue.batchStore(message, callback);//Generate the SequenceId value by using an auto-increment column.
timelineQueue.batchStore(sequenceId, message, callback);//Manually set the SequenceId value.
```

Get

You can call this operation to read a single row based on the SequenceId value. If no messages exist, no error occurs and the system returns an empty string.

```
timelineQueue.get(sequenceId);
```

GetLatestTimelineEntry

You can call this operation to read the latest message. If no messages exist, no error occurs and the system returns an empty string.

```
timelineQueue.getLatestTimelineEntry();
```

GetLatestSequenceId

You can call this operation to obtain the SequenceId value of the latest message. If no messages exist, no error occurs and the system returns 0.

```
timelineQueue.getLatestSequenceId();
```

Update

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

```
TimelineMessage message = new TimelineMessage().setField("text", "Timeline is fine.");
//update message with new field
message.setField("text", "new value");
timelineQueue.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>.

```
TimelineMessage oldMessage = new TimelineMessage().setField("text", "Timeline is fine.") ;
TimelineCallback callback = new TimelineCallback() {
    @Override
    public void onCompleted(TimelineIdentifier i, TimelineMessage m, TimelineEntry t) {
        // do something when succeed.
    }
    @Override
    public void onFailed(TimelineIdentifier i, TimelineMessage m, Exception e) {
        // do something when failed.
    }
};
TimelineMessage newMessage = oldMessage;
newMessage.setField("text", "new value");
timelineQueue.updateAsync(sequenceId, newMessage, callback);
```

Delete

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

```
timelineQueue.delete(sequenceId);
```

Scan

You can call this operation to read messages in a single Queue in order forwards or backwards based on the Scan parameter. This operation returns Iterator<TimelineEntry>. You can iterate all result sets by using the iterator.

```
ScanParameter scanParameter = new ScanParameter().scanBackward(Long.MAX_VALUE, 0);
timelineQueue.scan(scanParameter);
```

5.TimeSeries model

5.1. Overview

You can use the TimeSeries model to store, query, and analyze time series.

Public preview

The TimeSeries model is in public preview in the China (Shanghai), China (Hangzhou), Germany (Frankfurt), and China (Shenzhen) regions. During the public preview, you can use the TimeSeries model free of charge. To get started with the TimeSeries model, you can log on to the Tablestore console and click Create Public Preview Instance for TimeSeries Model to create a public preview instance for the TimeSeries model. For more information, see Create an instance for the TimeSeries model.

After you create a public preview instance for the TimeSeries model, you can use the Tablestore console, CLI, or SDKs to get started with the TimeSeries model. For more information, see Use the Tablestore console, Use the Tablestore CLI, and Use the SDK.

Background information

Tablestore is a multi-model data storage service that is developed by Alibaba Cloud. Tablestore can store large amounts of structured data and supports a variety of data models, including the TimeSeries model.

The TimeSeries model is designed based on the characteristics of time series data. This model is suitable for scenarios such as IoT device monitoring and can be used to store the data collected by devices and the monitoring data of machines. The TimeSeries model provides the following benefits:

- Provides a unified-common modeling method for time series data, which eliminates the need to predefine table schemas.
- Allows metadata indexes to be automatically created for time series and supports time series retrieval based on composite conditions.
- Supports queries and aggregation by using SQL.
- Supports automatic scale-out of service capabilities, high-concurrency writes and queries, and low-cost storage of petabytes of data.

Terms

Term	Description
Time series data	Time series data consists of multiple time series. Each time series is a set of data points that are arranged in chronological order. In addition, some metadata is needed to identify a time series. Therefore, time series data consists of metadata and data. ● Metadata: records the identifiers and properties of all time series. ● Data: records the data points of all time series. The data points include the time when the data points are generated and the corresponding data values.

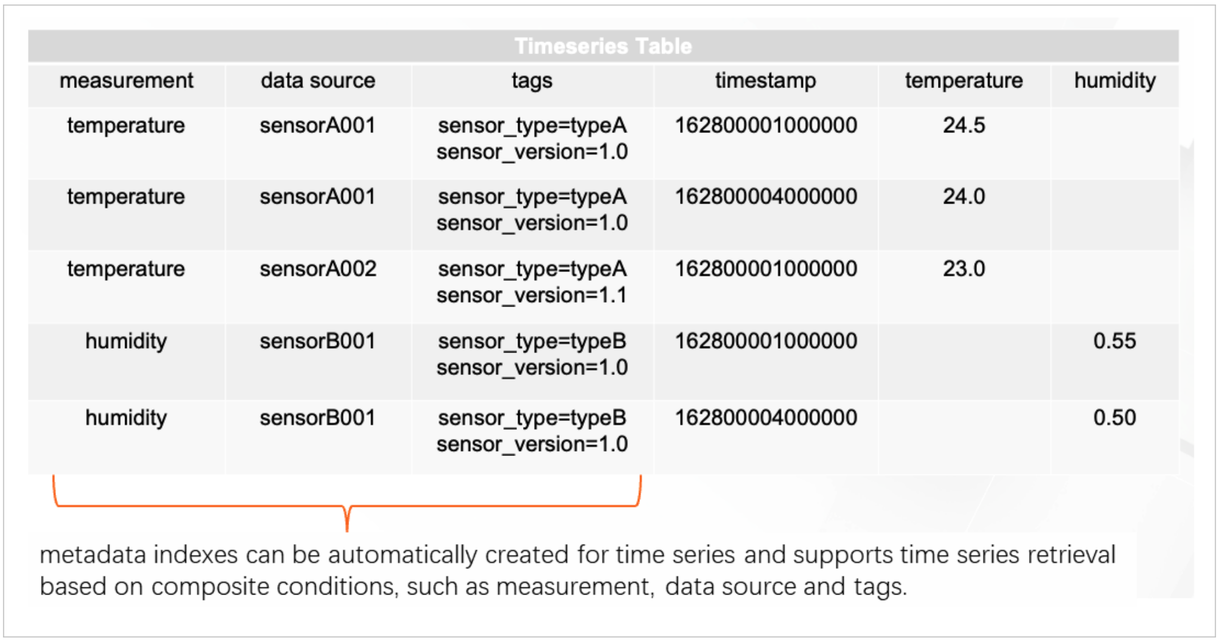
Term	Description
Time series metadata	Time series metadata contains the identifier and properties of a time series. The identifier is used to uniquely identify a time series. The properties can be modified and can be used for time series retrieval.
Time series identifier	A time series identifier is used to uniquely identify a time series. In the TimeSeries model of Tablestore, a time series identifier consists of the following three parts: metric name, data source, and tag.
Metric name	The name of a physical quantity or metric for data in a time series, such as cpu or net, which indicates that the CPU usage or network usage is recorded in the time series.
Data source	The identifier of the data source for the time series. This parameter can be empty.
Tag	The tag of the time series. You can customize multiple key-value pairs of the string type.
Property	Properties are part of time series metadata and can be used to record some modifiable property information of a time series. However, the properties cannot be used as the identifier of a time series and cannot be used to uniquely identify a time series. The properties of a time series are multiple key-value pairs of the string type, which are similar to tags in the format. You can specify or update the properties of a time series to retrieve the time series by using the properties.
Data in a time series	A data point in a time series consists of the time when the data is generated and the corresponding data value. If only one value is generated at each moment in a time series, the single-value model is used. If multiple values are generated at each moment in a time series, the multi-value model is used. The TimeSeries model of Tablestore uses the multi-value model. You can specify multiple data values at one point in time. Each value corresponds to a column in the time series table, including the column name and column value. Column values support the following data types: Boolean, integer, floating-point, string, and binary.

Data model

In the TimeSeries model of Tablestore, a two-dimensional time series table is used to store time series data.

Each row represents the data at a point in time in a time series. The time series identifier and timestamp are the primary key columns of the row, and the data points of the time series under the timestamp are the data columns of the row. A row can contain multiple data columns. You do not need to predefine the schemas of the primary key columns and data columns. You only need to specify the names of the specific data columns when you write data to the time series table.

A time series table can store time series data of different metric types. In the following figure, the time series table stores data of the following two metric types: temperature and humidity.



Timeseries Table

measurement	data source	tags	timestamp	temperature	humidity
temperature	sensorA001	sensor_type=typeA sensor_version=1.0	162800001000000	24.5	
temperature	sensorA001	sensor_type=typeA sensor_version=1.0	162800004000000	24.0	
temperature	sensorA002	sensor_type=typeA sensor_version=1.1	162800001000000	23.0	
humidity	sensorB001	sensor_type=typeB sensor_version=1.0	162800001000000		0.55
humidity	sensorB001	sensor_type=typeB sensor_version=1.0	162800004000000		0.50

metadata indexes can be automatically created for time series and supports time series retrieval based on composite conditions, such as measurement, data source and tags.

In the figure, the measurement, data source, and tags parameters form a time series identifier. You can use an API operation to update the properties in the metadata of a time series. The properties can be used to retrieve the time series.

After data is written to a time series table, the system automatically extracts the metadata of the time series and automatically creates a metadata index. You can retrieve a time series based on the combination of the metric name, data source, and tags.

Features

- Create and manage time series tables

  You can use the Tablestore console, SDKs, or CLI to query all time series tables in an instance, create a time series table, query the configurations of a time series table, update the configurations of a time series table, and delete a time series table.

  When you create a time series table or update the configurations of a time series table, you can specify the time to live (TTL) for the data in the time series table. After the TTL value is specified, the system automatically checks the difference between the current time and the timestamp of the time series data. If the difference exceeds the TTL value, the system automatically deletes the expired data.

- Read and write time series data

  You can use the Tablestore console, SDKs, or CLI to write multiple rows of time series data to a time series table at the same time. After data is written to the time series table, you can specify a time series identifier to query the data in the time series within the specified time range.

- Retrieve time series

  You can use the Tablestore console, SDKs, or CLI to retrieve time series in a time series table. You can use a composite condition that consists of multiple conditions to retrieve time series. For example, you can retrieve all time series in which the metric name is cpu, the tags contain a tag whose name is region and value is hangzhou, and the properties contain a property whose name is status and value is online. After the time series are retrieved, you can call an API operation to further query the data in the time series.

- Implement SQL query and analytics

Time series tables support queries by using SQL. In SQL, you can specify the metadata condition to filter time series and aggregate data based on the aggregation operations in different dimensions. For example, you can query the average value of the sample data from a batch of devices and aggregate second-level data into minute-level data.

In addition, you can query only the metadata of time series in SQL. This way, you can manage the metadata of time series by using SQL.

Limits

For more information, see Limits on the TimeSeries model.

5.2. Create an instance for the TimeSeries model

After you create an instance for the TimeSeries model, you can use the Tablestore console, CLI, or SDKs to get started with the TimeSeries model.

Procedure

1. Log on to the Tablestore console.
2. On the **Overview** page, click **Create Instance for TimeSeries Model**.
3. In the **Create Instance for TimeSeries Model** dialog box, select a region and specify Instance Name and Instance Description based on your business requirements.

   > **Notice** Each Alibaba Cloud account can create up to 10 instances. The name of an instance must be unique within the region in which the instance resides.

4. Click **OK**.

What's next

- Get started with the TimeSeries model by using the Tablestore console. For more information, see Use the Tablestore console.
- Get started with the TimeSeries model by using the Tablestore CLI. For more information, see Use the Tablestore CLI.
- Get started with the TimeSeries model by using Tablestore SDKs. For more information, see Use Tablestore SDKs.

5.3. Quick start

5.3.1. Use the Tablestore console

After you create a time series table in the console, you can write time series data to the time series table, and retrieve time series and query time series data in the time series table.

Prerequisites

A public preview instance for the TimeSeries model is created. For more information, see Create an instance for the TimeSeries model.

Step 1: Create a time series table

Create a time series table in the Tablestore console.

1. Log on to the Tablestore console.
2. On the **Overview** page, click the name of the instance in which you want to create a time series table or click **Manage Instance** in the Actions column that corresponds to the instance.
3. On the **Instance Details** tab, click the **Time Series Tables** tab.
4. On the **Time Series Tables** tab, click **Create Time Series Table**.

   > **Note** You can also click **Generate Demo with One Click** to create a test table with sample data for a quick start. When you create a time series table, the system performs some initialization operations. Therefore, you need to wait for dozens of seconds until the time series are displayed.

5. In the **Create Time Series Table** dialog box, specify **Name** and **Time to Live** as described in the following table.

Parameter	Description
Name	The name of the time series table, which is used to identify the time series table in an instance. The name must be 1 to 128 characters in length and can contain letters, digits, and underscores (_). The name must start with a letter or an underscore (_). The name of a time series table cannot be the same as the name of an existing data table.
Time to Live	The retention period of the data in the time series table. Unit: seconds. If the system detects that the difference between the current time and the time column that is passed to the table exceeds the specified TTL value, the system automatically deletes the expired data. **Notice** In the time series table, the system determines the time when the data is generated based on the time column that is passed to the table, not the time when the data is written to the table. The value of this parameter must be -1 or a value that is greater than or equal to 86400 seconds (one day).

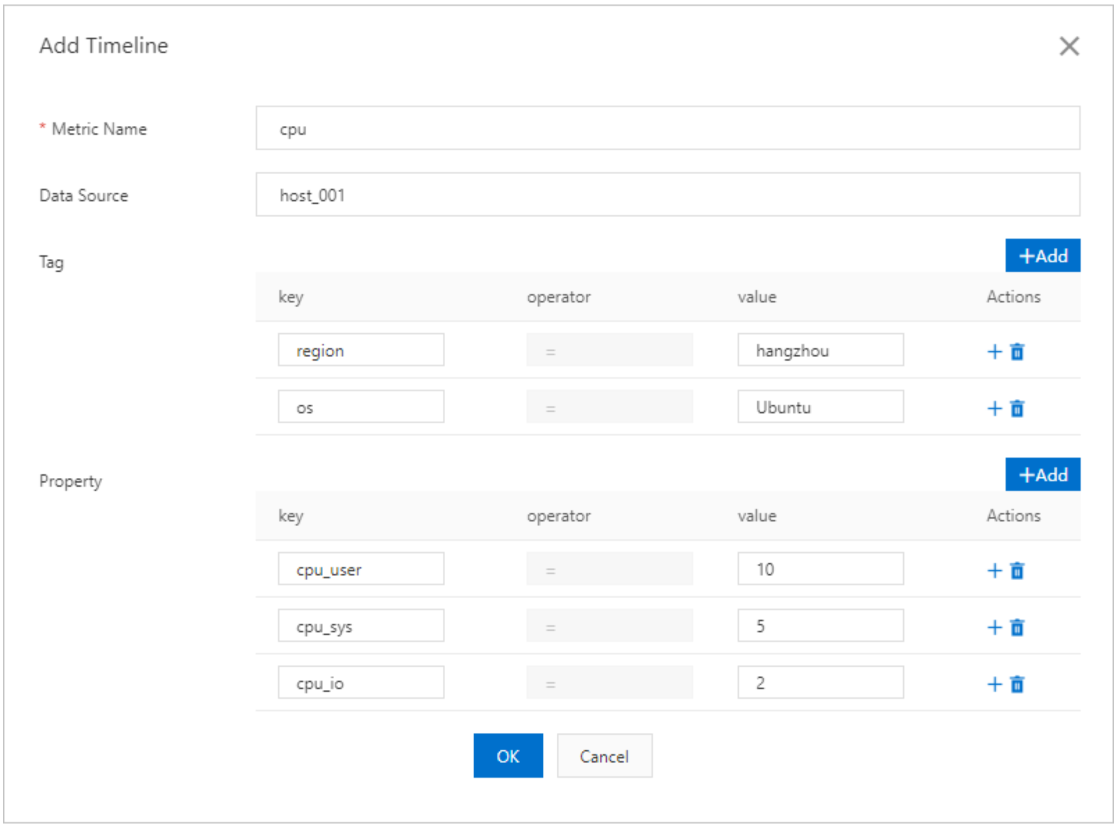
6. Click **OK**.

   After the time series table is created, you can view the time series table on the **Time Series Tables** tab. If the time series table is not displayed in the list of time series tables, click the ↻ icon to refresh the list of time series tables.

Step 2: Write time series data

Write time series data to the time series table in the Tablestore console. Time series data consists of metadata and data. If you do not create metadata before you write the time series data, the system automatically extracts the metadata from the written data.

1. On the **Time Series Tables** tab, click the name of the time series table and then click the **Query Data** tab or click **Manage Data** in the Actions column that corresponds to the time series table.
2. (Optional) Create a time series.
   i. On the **Query Data** tab, click **Add Timeline**.
   ii. In the **Add Timeline** dialog box, configure the metadata of the time series.



* Metric Name: cpu

Data Source: host_001

Tag

key	operator	value	Actions
region	=	hangzhou	+ 🗑
os	=	Ubuntu	+ 🗑

Property

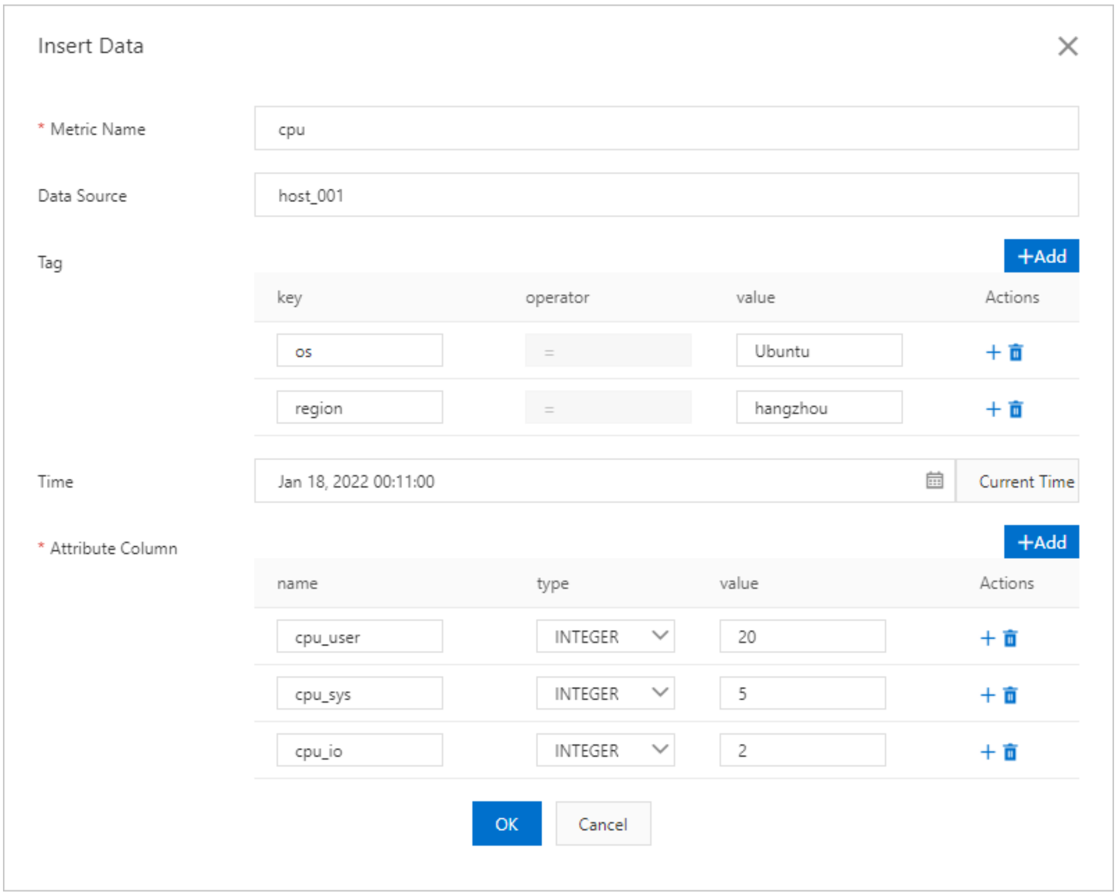
key	operator	value	Actions
cpu_user	=	10	+ 🗑
cpu_sys	=	5	+ 🗑
cpu_io	=	2	+ 🗑

The following table describes the parameters that you can configure to add a time series.

Parameter	Description
Metric Name	The name of a physical quantity or metric for the data in the time series, such as cpu or net, which specifies that the CPU usage or network usage is recorded in the time series.
Data Source	The identifier of the data source for the time series. This parameter can be empty.
Tag	The tag of the time series. You can customize multiple key-value pairs of the string type.
Property	The property column of the time series, which is used to record some property information of the time series.

   iii. Click **OK**.

3. Insert data.
   i. Click **Insert Data**.
   ii. In the **Insert Data** dialog box, specify Time and Attribute Column.



* Metric Name: cpu

Data Source: host_001

Tag

key	operator	value	Actions
os	=	Ubuntu	+ 🗑
region	=	hangzhou	+ 🗑

Time: Jan 18, 2022 00:11:00

* Attribute Column

name	type	value	Actions
cpu_user	INTEGER	20	+ 🗑
cpu_sys	INTEGER	5	+ 🗑
cpu_io	INTEGER	2	+ 🗑

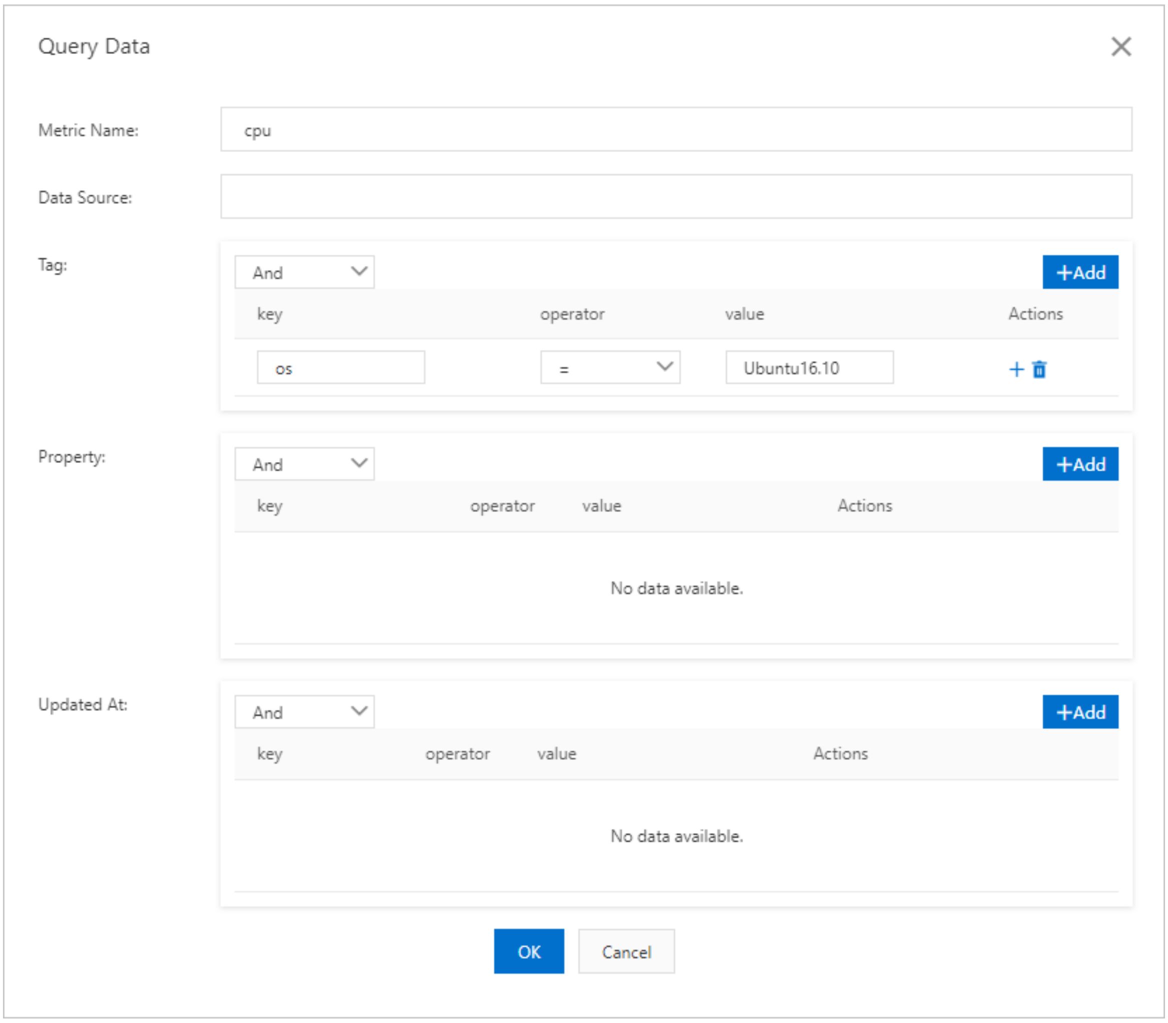
   iii. Click **OK**.

Step 3: Retrieve time series

Retrieves all the time series that meet the specified conditions.

1. On the **Query Data** tab, click **Query Data** in the upper-right corner.
2. In the **Query Data** dialog box, specify Metric Name and Data Source, and click **Add** in the Tag, Property, and Updated At sections to add conditions.

The following figure shows an example on how to query the time series in which the metric name is cpu and the tags contain os=Ubuntu16.10.



Query Data

Metric Name: cpu

Data Source:

Tag: And

key	operator	value	Actions
os	=	Ubuntu16.10	+ 🗑

Property: And

key	operator	value	Actions

No data available.

Updated At: And

key	operator	value	Actions

No data available.

3. Click **OK**.

   The time series that meet the conditions are displayed on the **Query Data** tab.

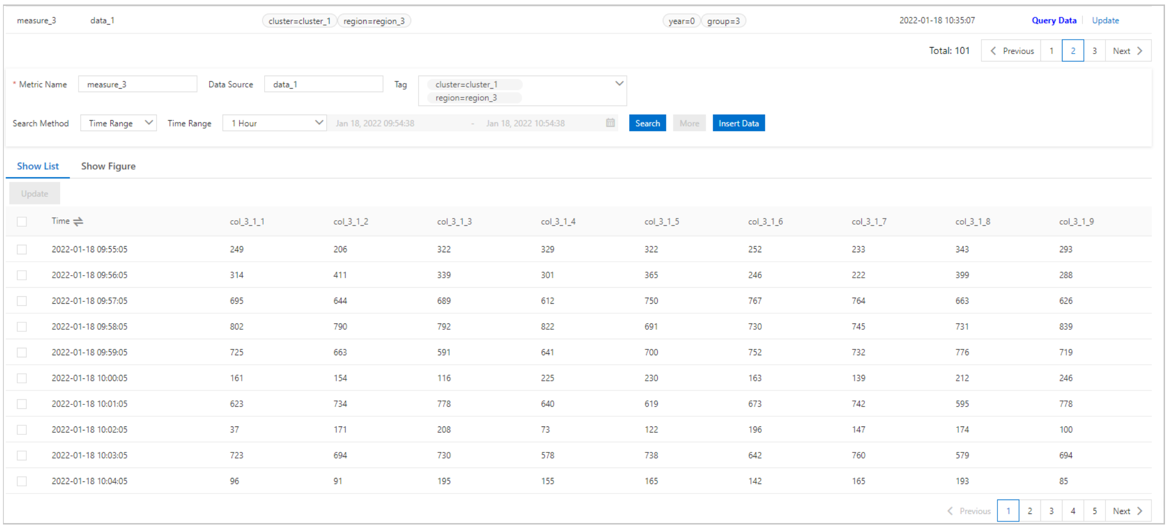
Step 4: Query time series data

Query the data in a time series within a specific time range.

1. On the **Query Data** tab, click **Query Data** in the **Actions** column that corresponds to the time series whose data you want to query.
2. Select Time Range or Microsecond Timestamp from the drop-down list of Search Method, specify the time, and click **Search**.

   The data that meets the conditions is displayed on the **Query Data** tab. The query results can be displayed in a list or figure.

The following figure shows an example of the query results in a list.



measure_3 data_1 cluster=cluster_1 region=region_3 year=0 group=3 2022-01-18 10:35:07 Query Data | Update

Total: 101 < Previous 1 2 3 Next >

* Metric Name: measure_3 Data Source: data_1 Tag: cluster=cluster_1 region=region_3

Search Method: Time Range Time Range: 1 Hour Jan 18, 2022 09:54:38 - Jan 18, 2022 10:54:38 Search More Insert Data

Show List Show Figure

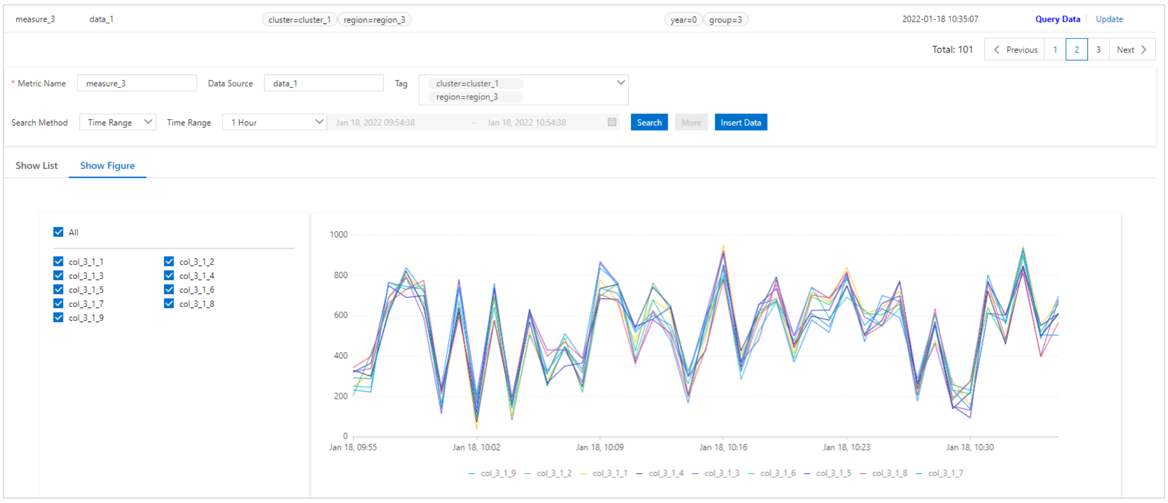
Update

□	Time ⇌	col_3_1_1	col_3_1_2	col_3_1_3	col_3_1_4	col_3_1_5	col_3_1_6	col_3_1_7	col_3_1_8	col_3_1_9
□	2022-01-18 09:55:05	249	206	322	329	322	252	233	343	293
□	2022-01-18 09:56:05	314	411	339	301	365	246	222	399	288
□	2022-01-18 09:57:05	695	644	689	612	750	767	764	663	626
□	2022-01-18 09:58:05	802	790	792	822	691	730	745	731	839
□	2022-01-18 09:59:05	725	663	591	641	700	752	732	776	719
□	2022-01-18 10:00:05	161	154	116	225	230	163	139	212	246
□	2022-01-18 10:01:05	623	734	778	640	619	673	742	595	778
□	2022-01-18 10:02:05	37	171	208	73	122	196	147	174	100
□	2022-01-18 10:03:05	723	694	730	578	738	642	760	579	694
□	2022-01-18 10:04:05	96	91	195	155	165	142	165	193	85

< Previous 1 2 3 4 5 Next >

The following figure shows an example of the query results in a figure.

Note Different colors in the figure represent different data columns. If you move the pointer over the data trend line, the values of the corresponding data columns are displayed. You can also select or clear specific data columns to display the required data columns.



measure_3 data_1 cluster=cluster_1 region=region_3 year=0 group=3 2022-01-18 10:35:07 Query Data | Update

Total: 101 < Previous 1 2 3 Next >

* Metric Name: measure_3 Data Source: data_1 Tag: cluster=cluster_1 region=region_3

Search Method: Time Range Time Range: 1 Hour Jan 18, 2022 09:54:38 - Jan 18, 2022 10:54:38 Search More Insert Data

Show List Show Figure

☑ All

☑ col_3_1_1 ☑ col_3_1_2

☑ col_3_1_3 ☑ col_3_1_4

☑ col_3_1_5 ☑ col_3_1_6

☑ col_3_1_7 ☑ col_3_1_8

☑ col_3_1_9

5.3.2. Use the Tablestore CLI

After you use the Tablestore CLI to create a time series table, you can run the CLI commands to write time series data to the time series table, and retrieve time series and query time series data in the time series table. You can also use SQL statements to retrieve time series and query time series data in the time series table.

Prerequisites

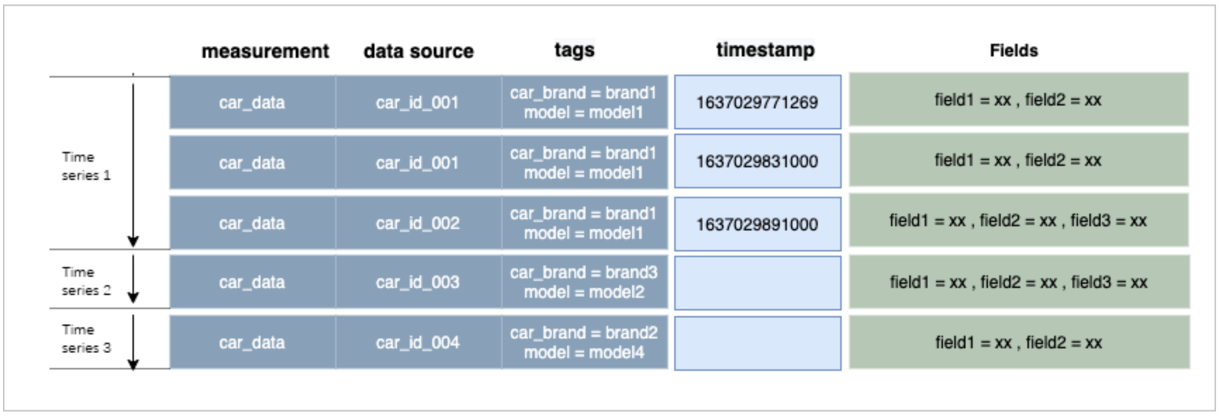
- A public preview instance for the TimeSeries model is created. For more information, see Create an instance for the TimeSeries model.
- The Tablestore CLI is downloaded. For more information, see Download the Tablestore CLI.
- The instance is started and configured. For more information, see Start the Tablestore CLI and

configure access information.

- An AccessKey pair is obtained. For more information, see Obtain an AccessKey pair.

Sample scenario

The following example shows how to use the TimeSeries model in the Internet of vehicles (IoV) scenario. In this example, a time series table named car_data is used. The table records the states of vehicles and contains the measurement, data source, tags, timestamp, and fields columns. The following figure shows the schema of the car_data time series table.



	measurement	data source	tags	timestamp	Fields
Time series 1	car_data	car_id_001	car_brand = brand1 model = model1	1637029771269	field1 = xx , field2 = xx
	car_data	car_id_001	car_brand = brand1 model = model1	1637029831000	field1 = xx , field2 = xx
	car_data	car_id_002	car_brand = brand1 model = model1	1637029891000	field1 = xx , field2 = xx , field3 = xx
Time series 2	car_data	car_id_003	car_brand = brand3 model = model2		field1 = xx , field2 = xx , field3 = xx
Time series 3	car_data	car_id_004	car_brand = brand2 model = model4		field1 = xx , field2 = xx

Operations on a time series table

1. Run the **create** command to create a time series table named car_data.

```
create -m timeseries -t car_data
```

2. Run the use --ts command to select the car_data time series table.

```
use --ts -t car_data
```

3. Import time series data by using one of the following methods:

   - Write a single row of time series data

     Run the putts command to write a single row of time series data. In the following example, a row of time series data is written to the car_data time series table.

```
putts --k '["car_data","car_0000010", ["brand=brand0","id=car_0000010","model=em3"]]'
--field '[{"c":"duration","v":121,"isint":true},{"c":"mileage","v":6480,"isint":true}
,{"c":"power","v":69,"isint":true},{"c":"speed","v":24,"isint":true},{"c":"temperatur
e","v":13,"isint":true}]' --time 1636460000000000
```

   - Import multiple rows of time series data at the same time

     Download the sample data and run the import_timeseries command to batch import the time series data. The sample data contains a total of 5 million rows of time series data. You can include the -l parameter in the command to specify the number of rows of time series data that you want to import. You can import up to 10 million rows of time series data free of charge.

     In the following example, 50,000 rows of time series data are imported. In the command, yourFilePath specifies the path where the sample data package is decompressed. Example: `D:\\timeseries_demo_data_5000000`.

```
import_timeseries -i yourFilePath -l 50000
```

Sample output:

```
Current speed is: 11000 rows/s. Total succeed count 11000, failed count 0.
Current speed is: 13000 rows/s. Total succeed count 24000, failed count 0.
Current speed is: 16400 rows/s. Total succeed count 40400, failed count 0.
Import finished, total count is 50000, failed 0 rows.
```

4. Run the qtm command to query time series. In the following example, all time series are queried and 10 time series are returned.

   ```
   qtm -l 10
   ```

   Sample output:

```
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| measurement | data_source | tags                                          | attribute
s | update_time      |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000005 | ["brand=brand0","id=car_0000005","model=m0"]  | null
| 1637722788684102 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000009 | ["brand=brand2","id=car_0000009","model=em3"] | null
| 1637722790158982 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000000 | ["brand=brand0","id=car_0000000","model=m3"]  | null
| 1637722787172818 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000008 | ["brand=brand0","id=car_0000008","model=m3"]  | null
| 1637722789832880 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000002 | ["brand=brand1","id=car_0000002","model=nm1"] | null
| 1637722787915852 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | null
| 1637722789006974 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000001 | ["brand=brand2","id=car_0000001","model=em2"] | null
| 1637722787260034 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000004 | ["brand=brand0","id=car_0000004","model=m2"]  | null
| 1637722788529313 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000003 | ["brand=brand1","id=car_0000003","model=nm0"] | null
| 1637722788288273 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
| car_data    | car_0000007 | ["brand=brand2","id=car_0000007","model=em2"] | null
| 1637722789315575 |
+-------------+-------------+-----------------------------------------------+----------
--+------------------+
```

5. Run the getts command to query the first five time points in a time series.

```
getts --k '["car_data","car_0000006", ["brand=brand2","id=car_0000006","model=em2"]]' -
l 5
```

Sample output:

```
+-------------+-------------+-----------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| measurement | data_source | tags                                          | timestamp
| duration | mileage | power | speed | temperature |
+-------------+-------------+-----------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656000
0000000 | 190      | 1770    | 33    | 54    | 29          |
+-------------+-------------+-----------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656001
0000000 | 554      | 6670    | 42    | 24    | 12          |
+-------------+-------------+-----------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656002
0000000 | 564      | 9750    | 14    | 75    | 22          |
+-------------+-------------+-----------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656003
0000000 | 176      | 7950    | 90    | 24    | 22          |
+-------------+-------------+-----------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656004
0000000 | 441      | 6280    | 30    | 38    | 31          |
+-------------+-------------+-----------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
```

Use SQL to query time series data

After you create a time series table in Tablestore, the system automatically creates two SQL mapping tables for the time series table: time series data table and time series metadata table.

- Time series data table: uses the same name as the time series table. You can query time series data in the time series data table. The name of the time series data table for the car_data time series table is `car_data`.
- Time series metadata table: uses a name that concatenates the `::meta` string after the name of the time series table. You can query time series metadata in the time series metadata table. The name of the time series metadata table for the car_data time series table is `car_data::meta`.

1. Run the sql command to enter the SQL mode.

   ```
   sql
   ```

2. Retrieve time series.

   - Example 1: Query the vehicles whose brand is brand0 and model is m3, and specify that only the first 10 query results are returned.

     ```
     SELECT * FROM `car_data::meta` WHERE _m_name = "car_data" AND tag_value_at(_tags,"bra
     nd") = "brand0" AND tag_value_at(_tags,"model") = "m3" LIMIT 10;
     ```

     Sample output:

```
+----------+--------------+----------------------------------------------+-----------
--+-------------------+
| _m_name  | _data_source | _tags                                        | _attribute
s | _meta_update_time |
+----------+--------------+----------------------------------------------+-----------
--+-------------------+
| car_data | car_0000000  | ["brand=brand0","id=car_0000000","model=m3"] | null
| 1637722787172818  |
+----------+--------------+----------------------------------------------+-----------
--+-------------------+
| car_data | car_0000008  | ["brand=brand0","id=car_0000008","model=m3"] | null
| 1637722789832880  |
+----------+--------------+----------------------------------------------+-----------
--+-------------------+
```

- Example 2: Query the total number of vehicles whose brand is brand2.

  ```
  SELECT count(*) FROM `car_data::meta` WHERE tag_value_at(_tags,"brand") = "brand2";
  ```

  Sample output:

  ```
  +----------+
  | count(*) |
  +----------+
  | 4        |
  +----------+
  ```

3. Query time series data.

   - Example 1: Query the vehicles whose metric name is car_data and data source is car_0000175, and specify that only the first 10 data points of the power data column are returned.

     ```
     SELECT _time, _field_name, _long_value as value FROM `car_data` WHERE _m_name = "car_
     data" AND _data_source = "car_0000001" AND _field_name = "power" LIMIT 10;
     ```

     Sample output:

```
+------------------+------------+-------+
| _time            | _field_name | value |
+------------------+------------+-------+
| 1636560000000000 | power      | 68    |
+------------------+------------+-------+
| 1636560010000000 | power      | 41    |
+------------------+------------+-------+
| 1636560020000000 | power      | 69    |
+------------------+------------+-------+
| 1636560030000000 | power      | 95    |
+------------------+------------+-------+
| 1636560040000000 | power      | 27    |
+------------------+------------+-------+
| 1636560050000000 | power      | 26    |
+------------------+------------+-------+
| 1636560060000000 | power      | 98    |
+------------------+------------+-------+
| 1636560070000000 | power      | 82    |
+------------------+------------+-------+
| 1636560080000000 | power      | 24    |
+------------------+------------+-------+
| 1636560090000000 | power      | 2     |
+------------------+------------+-------+
```

- Example 2: Query the maximum speed of vehicles whose metric name is car_data and data source is car_000002.

  ```
  SELECT max(_long_value) as speed FROM `car_data` WHERE _m_name = "car_data" AND _data
  _source = "car_0000002" AND _field_name = "speed";
  ```

  Sample output:

  ```
  +--------+
  | speed |
  +--------+
  | 100    |
  +--------+
  ```

- Example 3: Aggregate the temperature data of vehicles whose metric name is car_data and data source is car_0000001 based on a time window of 60s to calculate the lowest temperature per minute.

  ```
  SELECT _time DIV 60000000 * 60 as time_sec, min(_long_value) as temperature FROM `car
  _data` WHERE _data_source = "car_0000001" AND _field_name = "temperature" GROUP BY ti
  me_sec ORDER BY time_sec ASC LIMIT 10;
  ```

  Sample output:

```
+------------+-------------+
| time_sec   | temperature |
+------------+-------------+
| 1636560000 | 11          |
+------------+-------------+
| 1636560060 | 10          |
+------------+-------------+
| 1636560120 | 11          |
+------------+-------------+
| 1636560180 | 10          |
+------------+-------------+
| 1636560240 | 11          |
+------------+-------------+
| 1636560300 | 12          |
+------------+-------------+
| 1636560360 | 14          |
+------------+-------------+
| 1636560420 | 10          |
+------------+-------------+
| 1636560480 | 15          |
+------------+-------------+
| 1636560540 | 11          |
+------------+-------------+
```

4. Exit the SQL mode

```
exit;
```

5. Exit the Tablestore CLI.

```
exit
```

5.4. Use Tablestore SDKs

After you use a Tablestore SDK to create a time series table, you can write time series data to the time series table, and retrieve time series and query time series data in the time series table.

Operations

Operation	Description
CreateTimeseriesTable	Creates a time series table.
ListTimeseriesTable	Queries the time series tables in the current instance.
DescribeTimeseriesTable	Queries the information about a time series table.
UpdateTimeseriesTable	Updates the configurations of a time series table.
DeleteTimeseriesTable	Deletes a time series table.
PutTimeseriesData	Writes time series data.

Operation	Description
GetTimeseriesData	Queries the data in a time series.
QueryTimeseriesMeta	Retrieves the metadata of a time series.
UpdateTimeseriesMeta	Updates the metadata of a time series.
DeleteTimeseriesMeta	Deletes the metadata of a time series.

Tablestore SDKs

You can use the following Tablestore SDKs to get started with the TimeSeries model:

- Java SDK
- Go SDK

Create a time series table

You can call the CreateTimeseriesTable operation to create a time series table. When you call the CreateTimeseriesTable operation to create a time series table, you must specify the configurations of the table.

- Parameters

  The schema information (timeseriesTableMeta) about a time series table includes the name (timeseriesTableName) and configurations (timeseriesTableOptions) of the table. The following table describes the parameters.

Parameter	Description
timeseriesTableName	The name of the time series table.
timeseriesTableOptions	The configurations of the time series table. The configurations include the following content: timeToLive: the retention period of the data in the time series table. Unit: seconds. If you want the data in the time series table to never expire, set this parameter to -1. You can call the UpdateTimeseriesTable operation to change the value of this parameter.

- Examples

  Create a time series table named test_timeseries_table in which the data never expires.

```
private static void createTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    TimeseriesTableMeta timeseriesTableMeta = new TimeseriesTableMeta(tableName);
    int timeToLive = -1;
    timeseriesTableMeta.setTimeseriesTableOptions(new TimeseriesTableOptions(timeToLive))
;
    CreateTimeseriesTableRequest request = new CreateTimeseriesTableRequest(timeseriesTab
leMeta);
    client.createTimeseriesTable(request);
}
```

Write time series data

You can call the PutTimeseriesData operation to write multiple rows of time series data at a time.

- Parameters

  A row of time series data (timeseriesRow) includes the time series identifier (timeseriesKey) and time series data. The time series data includes data points (fields) and the time (timeInUs) of the data points. The following table describes the parameters.

Parameter	Description
timeseriesKey	The identifier of the time series. The identifier includes the following content: ○ measurementName: the measurement name of the time series. ○ dataSource: the data source of the time series. You can leave this parameter empty. ○ tags: the tags of the time series. The tags are multiple key-value pairs of the STRING type.
timeInUs	The time of the data point. Unit: microseconds.
fields	The data points, which can be multiple pairs of names (FieldKey) and data values (FieldValue).

- Examples

  Write multiple rows of time series data to the time series table named test_timeseries_table at a time.

```
private static void putTimeseriesData(TimeseriesClient client) {
    List<TimeseriesRow> rows = new ArrayList<TimeseriesRow>();
    for (int i = 0; i < 10; i++) {
        Map<String, String> tags = new HashMap<String, String>();
        tags.put("region", "hangzhou");
        tags.put("os", "Ubuntu16.04");
        // Use the measurement name, data source, and tags of a time series to construct the identifier of the time series.
        TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_" + i, tags);
        // Specify the timeseriesKey and timeInUs parameters to create a row of time series data.
        TimeseriesRow row = new TimeseriesRow(timeseriesKey, System.currentTimeMillis() * 1000 + i);
        // Add data values (fields).
        row.addField("cpu_usage", ColumnValue.fromDouble(10.0));
        row.addField("cpu_sys", ColumnValue.fromDouble(5.0));
        rows.add(row);
    }
    String tableName = "test_timeseries_table";
    PutTimeseriesDataRequest putTimeseriesDataRequest = new PutTimeseriesDataRequest(tableName);
    putTimeseriesDataRequest.setRows(rows);
    // Write multiple rows of time series data at a time.
    PutTimeseriesDataResponse putTimeseriesDataResponse = client.putTimeseriesData(putTimeseriesDataRequest);
    // Check whether all data is written to the time series table.
    if (!putTimeseriesDataResponse.isAllSuccess()) {
        for (PutTimeseriesDataResponse.FailedRowResult failedRowResult : putTimeseriesDataResponse.getFailedRows()) {
            System.out.println(failedRowResult.getIndex());
            System.out.println(failedRowResult.getError());
        }
    }
}
```

Retrieve time series

- Parameters

  The metaQueryCondition parameter specifies the conditions for a time series retrieval. The conditions include compositeMetaQueryCondition, measurementMetaQueryCondition, dataSourceMetaQueryCondition, tagMetaQueryCondition, attributeMetaQueryCondition, and updateTimeMetaQueryCondition. The following table describes the parameters.

Parameter	Description
compositeMetaQueryCondition	The composite condition that includes the following content: ◦ operator: the logical operator. Valid values: AND, OR, and NOT. ◦ subConditions: the subconditions that can be combined by using operators for complex queries.

Parameter	Description
measurementMetaQueryCondition	The measurement name condition that includes the following content: ○ operator: the relational operator or the prefix match condition. Valid values for the relational operator: =, !=, >, >=, <, and <=. ○ value: the measurement name of the time series that you want to retrieve. Type: STRING.
dataSourceMetaQueryCondition	The data source condition that includes the following content: ○ operator: the relational operator or the prefix match condition. Valid values for the relational operator: =, !=, >, >=, <, and <=. ○ value: the data source of the time series that you want to retrieve. Type: STRING.
tagMetaQueryCondition	The tag condition that includes the following content: ○ operator: the relational operator or the prefix match condition. Valid values for the relational operator: =, !=, >, >=, <, and <=. ○ value: the tag of the time series that you want to retrieve. Type: STRING.
attributeMetaQueryCondition	The attribute condition for the metadata of the time series. The attribute condition includes the following content: ○ operator: the relational operator or the prefix match condition. Valid values for the relational operator: =, !=, >, >=, <, and <=. ○ attributeName: the name of the attribute. Type: STRING. ○ value: the value of the attribute. Type: STRING.
updateTimeMetaQueryCondition	The update time condition for the metadata of the time series. The update time condition includes the following content: ○ operator: the relational operator. Valid values: =, !=, >, >=, <, and <=. ○ timeInUs: the timestamp when the metadata of the time series is updated. Unit: microseconds.

- Examples

  Query all time series in which the measurement name is cpu and the tags include the os tag whose tag value is prefixed with Ubuntu in the time series table named test_timeseries_table.

```
private static void queryTimeseriesMeta(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    QueryTimeseriesMetaRequest queryTimeseriesMetaRequest = new QueryTimeseriesMetaRequest(tableName);
    // Query all time series in which the measurement name is cpu and the tags include the os tag whose tag value is prefixed with Ubuntu. Condition: measurement_name="cpu" and have_prefix(os, "Ubuntu").
    CompositeMetaQueryCondition compositeMetaQueryCondition = new CompositeMetaQueryCondition(MetaQueryCompositeOperator.OP_AND);
    compositeMetaQueryCondition.addSubCondition(new MeasurementMetaQueryCondition(MetaQuerySingleOperator.OP_EQUAL, "cpu"));
    compositeMetaQueryCondition.addSubCondition(new TagMetaQueryCondition(MetaQuerySingleOperator.OP_PREFIX, "os", "Ubuntu"));
    queryTimeseriesMetaRequest.setCondition(compositeMetaQueryCondition);
    queryTimeseriesMetaRequest.setGetTotalHits(true);
    QueryTimeseriesMetaResponse queryTimeseriesMetaResponse = client.queryTimeseriesMeta(queryTimeseriesMetaRequest);
    System.out.println(queryTimeseriesMetaResponse.getTotalHits());
    for (TimeseriesMeta timeseriesMeta : queryTimeseriesMetaResponse.getTimeseriesMetas()) {
        System.out.println(timeseriesMeta.getTimeseriesKey().getMeasurementName());
        System.out.println(timeseriesMeta.getTimeseriesKey().getDataSource());
        System.out.println(timeseriesMeta.getTimeseriesKey().getTags());
        System.out.println(timeseriesMeta.getAttributes());
        System.out.println(timeseriesMeta.getUpdateTimeInUs());
    }
}
```

Query time series data

- Parameters

Parameter	Description
timeseriesKey	The identifier of the time series that you want to query. The identifier includes the following content: ○ measurementName: the measurement name of the time series. ○ dataSource: the data source of the time series. You can leave this parameter empty. ○ tags: the tags of the time series. The tags are multiple key-value pairs of the STRING type.
timeRange	The time range for the query. The time range is a left-open, right-closed interval. The time range includes the following content: ○ beginTimeInUs: the start time. ○ endTimeInUs: the end time.

Parameter	Description
backward	Specifies whether to sort the query results in reverse chronological order. This allows you to obtain the latest data in a time series. Valid values: ○ true: sorts the query results in reverse chronological order. ○ false: sorts the query results in chronological order. This is the default value.
fieldsToGet	The columns that you want to return. If you do not specify this parameter, all columns are returned. **Notice** When you specify the fieldsToGet parameter, you must specify the name and data type of each column that you want to return. If the specified data type of a column is not that of the column in the time series table, the data of the column cannot be returned.
limit	The maximum number of rows that you want to return. **Note** The limit parameter limits only the maximum number of rows that you want to return. Even if the number of rows that meet the specified conditions exceeds the limit, the number of rows that are returned may be less than the value of the limit parameter due to other limits such as the maximum amount of data for a scan. In this case, you can obtain the remaining rows by using the nextToken parameter.
nextToken	The token that is used to obtain more results. If only some rows that meet the specified conditions are returned in a query, the response contains the nextToken parameter. You can specify the nextToken parameter in the next request to obtain the remaining rows.

- Examples

Query the time series data that meets the specified conditions in the time series table named test_timeseries_table.

```
private static void getTimeseriesData(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    GetTimeseriesDataRequest getTimeseriesDataRequest = new GetTimeseriesDataRequest(tableName);
    Map<String, String> tags = new HashMap<String, String>();
    tags.put("region", "hangzhou");
    tags.put("os", "Ubuntu16.04");
    // Use the measurement name, data source, and tags of a time series to construct the identifier of the time series.
    TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_0", tags);
    getTimeseriesDataRequest.setTimeseriesKey(timeseriesKey);
    // Specify the time range.
    getTimeseriesDataRequest.setTimeRange(0, (System.currentTimeMillis() + 60 * 1000) * 1000);
    // Specify the maximum number of rows that you want to return.
    getTimeseriesDataRequest.setLimit(10);
    // Optional. Specify whether to sort the query results in reverse chronological order. Default value: false. If you set this parameter to true, the query results are sorted in reverse chronological order.
    getTimeseriesDataRequest.setBackward(false);
    // Optional. Specify the columns that you want to return. If you do not specify this parameter, all columns are returned.
    getTimeseriesDataRequest.addFieldToGet("string_1", ColumnType.STRING);
    getTimeseriesDataRequest.addFieldToGet("long_1", ColumnType.INTEGER);
    GetTimeseriesDataResponse getTimeseriesDataResponse = client.getTimeseriesData(getTimeseriesDataRequest);
    System.out.println(getTimeseriesDataResponse.getRows().size());
    if (getTimeseriesDataResponse.getNextToken() != null) {
        // If the nextToken parameter is not empty, you can initiate a request again to obtain the remaining data.
        getTimeseriesDataRequest.setNextToken(getTimeseriesDataResponse.getNextToken());
        getTimeseriesDataResponse = client.getTimeseriesData(getTimeseriesDataRequest);
        System.out.println(getTimeseriesDataResponse.getRows().size());
    }
}
```

5.5. Use SQL to query time series data

After you create a time series table in Tablestore and a mapping table for the time series table in SQL, you can execute SQL statements in the Tablestore console or by using Tablestore SDKs to query time series data in the time series table.

Mapping tables for a time series table in SQL

The time series model is classified into the single-value model and the multi-value model based on whether one or more values are generated at each time point in a time series. The following table describes the types of mapping tables that you can create for a time series table in SQL to query data.

Mapping table type	Description	Creation method	Name of the mapping table in SQL
Mapping tables in the single-value model	Queries time series data by using the mapping table in the single-value model.	After you create a time series table, the system automatically creates a mapping table in SQL for the time series table.	Same as the name of the time series table.
Mapping tables in the multi-value model	Queries time series data by using the mapping table in the multi-value model.	After you create a time series table, you manually create a mapping table in SQL.	The name of the mapping table is in the `Name of the time series table::Suffix` format. Specify `Suffix` when you create a mapping table in SQL.
Mapping tables for time series metadata	Queries time series metadata.	After you create a time series table, the system automatically creates a mapping table in SQL.	The name of the mapping table is in the `Name of the time series table::meta` format.

Mapping tables in the single-value model

After you create a time series table, the system automatically creates a mapping table in the single-value model in SQL for the time series table. The name of the mapping table in SQL is the same as the name of the time series table. You can use the mapping table in the single-value model to query time series data in the time series table.

The following table describes the schema of the mapping table in SQL.

Column name	Type	Description
_m_name	VARCHAR	The metric name.
_data_source	VARCHAR	The data source.
_tags	VARCHAR	The tags of the time series. The value is an array and multiple tags are in the ["tagKey1=tagValue1","tagKey2=tagValue2"] format. You can use the tag_value_at function to extract the value of a tag.
_time	BIGINT	The timestamp of the data point. Unit: microseconds.
_field_name	VARCHAR	The name of the data column.

Column name	Type	Description
_long_value	BIGINT	The value of the integer type. If the data type of the data column is not integer, the value is NULL.
_double_value	DOUBLE	The value of the floating-point type. If the data type of the data column is not floating-point, the value is NULL.
_bool_value	BOOL	The value of the Boolean type. If the data type of the data column is not Boolean, the value is NULL.
_string_value	VARCHAR	The value of the string type. If the data type of the data column is not string, the value is NULL.
_binary_value	MEDIUMBLOB	The value of the binary type. If the data type of the data column is not binary, the value is NULL.
_attributes	VARCHAR	The properties of the time series. The format of properties is the same as the format of tags.
_meta_update_time	BIGINT	The point in time when the metadata of the time series is updated. When you update the properties of a time series, the system automatically updates the metadata update time of the time series. If you continue to write data to the time series, the system updates the metadata update time of the time series at regular intervals. You can use the metadata update time to determine whether the time series is active.

Mapping tables in the multi-value model

If you want to query time series data by using a mapping table in the multi-value model, execute the CREATE TABLE statement to create a mapping table in the multi-value model. The mapping table in SQL uses a name that concatenates the `::Suffix` string to the name of the time series table. Specify `Suffix` when you create a mapping table in SQL. You can create multiple mapping tables in the multi-value model in SQL for a time series table.

When you create a mapping table in the multi-value model for a time series table, specify the name of the mapping table, and the names and types of the data columns in the mapping table. For more information, see Create mapping tables in the multi-value model for time series tables.

The following table describes the schema of the mapping table in SQL.

Note If you want to read the properties column (_attributes) of the time series metadata or the metadata update time column (_meta_update_time) by using a mapping table in the multi-value model, add the two columns to the mapping table. The system automatically fills the content in the two metadata columns.

Column name	Type	Description
_m_name	VARCHAR	The metric name.
_data_source	VARCHAR	The data source.
_tags	VARCHAR	The tags of the time series. The value is an array and multiple tags are in the ["tagKey1=tagValue1","tagKey2=tagValue2"] format. You can use the tag_value_at function to extract the value of a tag.
_time	BIGINT	The timestamp of the data point. Unit: microseconds.
The name of the custom data column.	SQL data types	You can add multiple custom data columns to the mapping table in SQL. If the name or type of the specified column in the mapping table in SQL does not match the name or type of the column in the time series table, the values of the column in the mapping table are null.
_attributes (optional)	MEDIUMTEXT	The properties of the time series. The format of properties is the same as the format of tags.
_meta_update_time (optional)	BIGINT	The point in time when the metadata of the time series is updated. When you update the properties of a time series, the system automatically updates the metadata update time of the time series. If you continue to write data to the time series, the system updates the metadata update time of the time series at regular intervals. You can use the metadata update time to determine whether the time series is active.

Mapping tables for time series metadata

After you create a time series table, the system automatically creates a mapping table for time series metadata. The mapping table uses a name that concatenates the `::meta` string to the name of the time series table. You can use the mapping table to query time series metadata. For example, if the name of the time series table is timeseries_table, the name of the mapping table for time series metadata is `timeseries_table::meta`.

The following table describes the schema of the mapping table in SQL.

Column name	Type	Description
_m_name	VARCHAR	The metric name.
_data_source	VARCHAR	The data source.
_tags	VARCHAR	The tags of the time series.

Column name	Type	Description
_attributes	VARCHAR	The properties of the time series.
_meta_update_time	BIGINT	The point in time when the metadata of the time series is updated. When you update the properties of a time series, the system automatically updates the metadata update time of the time series. If you continue to write data to the time series, the system updates the metadata update time of the time series at regular intervals. You can use the metadata update time to determine whether the time series is active.

SQL syntax

Create mapping tables in the multi-value model for time series tables

You can execute the CREATE TABLE statement to create a mapping table in the multi-value model for a time series table.

- SQL syntax

  ```
  CREATE TABLE `timeseries_table::user_mapping_name` (
    `_m_name` VARCHAR(1024),
    `_data_source` VARCHAR(1024),
    `_tags` VARCHAR(1024),
    `_time` BIGINT(20),
    `user_column_name1 ` data_type,
    ......
    `user_column_namen ` data_type,
    PRIMARY KEY(`_m_name`,`_data_source`,`_tags`,`_time`)
  );
  ```

  For more information about the parameters in the SQL syntax, see the table schema in Mapping tables in the multi-value model.

- SQL example

  The following sample code shows how to create a mapping table in the multi-value model named `timeseries_table::muti_model` for the time series table. The types of metrics in the mapping table are cpu, memory, and disktop. SQL sample code:

```
CREATE TABLE `timeseries_table::muti_model` (
  `_m_name` VARCHAR(1024),
  `_data_source` VARCHAR(1024),
  `_tags` VARCHAR(1024),
  `_time` BIGINT(20),
  `cpu` DOUBLE(10),
  `memory` DOUBLE(10),
  `disktop` DOUBLE(10),
  PRIMARY KEY(`_m_name`,`_data_source`,`_tags`,`_time`)
);
```

Query data

You can execute the SELECT statement to query time series data. For more information, see Query data.

Tablestore provides the tag_value_at extension function to allow you to extract the value of a tag in the tags (_tags) of a time series. You can also use the function to extract the value of a property in the properties (_attributes) of a time series.

If the value of _tags is ["host=abc","region=hangzhou"], you can use tag_value_at(_tags, "host") to extract the value abc of the host tag. The following SQL statement shows an example:

```
SELECT tag_value_at(_tags, "host") as host FROM timeseries_table LIMIT 1;
```

SQL examples

Query time series

After you create a time series table, the system automatically creates a mapping table for time series metadata. You can use the mapping table to query time series.

In this example, a time series table named timeseries_table and a mapping table for time series metadata named `timeseries_table::meta` are used. The type of the metric in the mapping table is basic_metric.

- Query time series whose metric type is basic_metric in the time series metadata table.

```
SELECT * FROM  `timeseries_table::meta` WHERE _m_name = "basic_metric" LIMI 100;
```

- Query time series that meet multiple tag conditions (host=host001 and region=hangzhou) in the time series metadata table.

```
SELECT * FROM `timeseries_table::meta` WHERE _m_name = "basic_metric" AND tag_value_at(_t
ags, "host") = "host001"
 AND tag_value_at(_tags, "region") = "hangzhou" LIMI 100;
```

Query time series data by using the mapping table in the single-value model

After you create a time series table, the system automatically creates a mapping table in the single-value model for the time series table. You can use the mapping table to query time series data.

In this example, a time series table named timeseries_table and a mapping table in the single-value model named `timeseries_table` are used. The type of the metric in the mapping table is basic_metric.

- Query the data whose metric type is basic_metric in the time series data table.

```
SELECT * FROM timeseries_table WHERE _m_name = "basic_metric" LIMIT 10;
```

- Query the data in the time series that meets a single tag condition (host=host001) in the time series data table.

```
SELECT * FROM timeseries_table WHERE _m_name = "basic_metric" AND tag_value_at(_tags, "ho
st") = "host001"
 AND _time > (UNIX_TIMESTAMP() - 900) * 1000000 LIMIT 10;
```

- Query the data in the time series that meets multiple tag conditions (host=host001 and region=hangzhou) in the time series data table.

```
SELECT * FROM timeseries_table WHERE _m_name = "basic_metric" AND tag_value_at(_tags, "ho
st") = "host001" AND tag_value_at(_tags, "region") = "hangzhou"
 AND _time > (UNIX_TIMESTAMP() - 900) * 1000000 LIMIT 10;
```

Query time series data by using a mapping table in the multi-value model

After you create a time series table, you can create a mapping table in the multi-value model for the time series table. You can use the mapping table to query time series data. For more information about how to create a mapping table in the multi-value model for a time series table, see Create mapping tables in the multi-value model for time series tables.

In this example, a time series table named timeseries_table and a mapping table in the multi-value model named `timeseries_table::muti_model` are used. The types of metrics in the mapping table are cpu, memory, and disktop.

- Query the data whose data source is host_01 by using the mapping table in the multi-value model. In this example, host_id is stored in _data_source.

```
SELECT * FROM `timeseries_table::muti_model` WHERE _data_source = "host_01" LIMIT 10;
```

- Query information about the metrics in the time series whose cpu value is greater than 20.0 by using the mapping table in the multi-value model.

```
SELECT cpu,memory,disktop FROM `timeseries_table::muti_model` WHERE cpu > 20.0 LIMIT 10;
```

- Calculate the average cpu values and maximum disktop values of the hosts that meet a specific tag condition (region=hangzhou) on January 1, 2022 by using the mapping table in the multi-value model.

```
SELECT avg(cpu) as avg_cpu,max(disktop) as max_disktop FROM `timeseries_table::muti_mode`
WHERE tag_value_at(_tags,"region") = "hangzhou"
 AND _time > 1640966400000000 AND _time < 1641052799000000 GROUP BY _data_source;
```

Methods

You can use SQL to query time series data by using one of the following methods. When you query time series data, you can perform operations on the mapping tables based on your business requirements.

- Use SQL to query time series data in the Tablestore console. For more information, see Use the Tablestore console.
- Use SQL to query time series data by using the Tablestore SDKs. For more information, see Use Tablestore SDKs.

- Use SQL to query time series data by using Java Database Connectivity (JDBC). For more information, see Use JDBC to access Tablestore.
- Use SQL to query time series data in the Tablestore CLI. For more information, see SQL query.

6.Search Index

6.1. Overview

The search index feature provides multiple efficient index schemas to help you process complex queries in big data scenarios.

Purposes

Data tables in Tablestore use distributed NoSQL data structures. Data such as monitoring data and log data can be stored, read, and written at a large scale.

In addition to queries based on primary keys including single-row read and range read, Tablestore provides the search index feature to meet your requirements for complex queries. These queries include Boolean query and queries based on non-primary key columns.

The search index feature is implemented by using inverted indexes and column stores. This feature provides query methods to solve problems in complex big data scenarios. The query methods include queries based on non-primary key columns, full-text search, prefix query, fuzzy query, Boolean query, nested query, and geo query. Aggregation can be implemented by using max, min, count, sum, avg, distinct_count, and group_by.

Differences among indexes

Aside from queries based on primary keys in data tables, Tablestore provides two index schemas for accelerated queries: secondary index and search index. The following table describes the differences among the three types of indexes.

Index type	Description	Scenario
Primary key of a data table	A data table is similar to a large map. Data tables support queries based only on primary keys.	You can specify a complete primary key or the prefix of a primary key.
Secondary index	You can create one or more index tables and perform queries by using the primary key columns of the index tables.	You can define the required columns in advance. Therefore, only a small number of columns are queried. You can also specify a complete primary key or the prefix of a primary key.

Index type	Description	Scenario
Search index	The search index feature uses inverted indexes, Bkd-trees, and column stores for various query scenarios.	The following scenarios do not support the use of a primary key of a data table and the secondary index feature: ● Query based on non-primary key columns ● Boolean query ● Query by using operators such as AND, OR, and NOT ● Full-text search ● Geo query ● Prefix query ● Fuzzy query ● Nested query ● Exists query ● Aggregation by using min, max, sum, avg, count, distinct_count, and group_by

Compared with indexes of conventional database services such as MySQL, the search index feature is not subject to the leftmost matching principle. Therefore, the search index feature can be used in more scenarios. In most cases, only one search index is required for a data table. For example, a data table about student information may contain the student name, ID, gender, grade, class, and home address columns. When you create a search index, you can add these columns to the search index. When you use the search index, you can specify a combination of conditions. Examples: students named Tom in Grade Three, male students who live one kilometer away from their school, and students in Class Two, Grade Three who live in the specified residential community.

API operations

The search index feature provides the Search and ParallelScan API operations. The Search API operation is used for general queries. The ParallelScan API operation is used for data export.

Most features that are provided by the two API operations are the same. However, to improve the performance and throughput, the ParallelScan API operation does not provide some features of the Search API operation. The following table compares the two API operations.

API operation	Description
Search	Supports all features of search indexes. ● Query: query based on non-primary key columns, full-text search, prefix query, fuzzy query, Boolean query, nested query, and geo query. ● Collapse (distinct) ● Sorting ● Aggregation ● Total number of rows

API operation	Description
ComputeSplits+ParallelScan	Exports data in parallel. The query feature of search indexes is supported. However, analysis features such as sorting and aggregation are not supported. The query speed of a ParallelScan request that includes a parallel scan task is five times faster than the query speed of a Search request. ● Query: query based on non-primary key columns, full-text search, prefix query, fuzzy query, Boolean query, nested query, and geo query. ● Multiple parallel scan tasks included in one ParallelScan request

Usage notes

Notice Predefined columns are not required when you use a search index.

- Index synchronization

  After a search index is created for a data table, data is written to the data table first. After the data is written to the data table, a success message is returned. At the same time, another asynchronous thread reads the newly written data from the data table and writes the data to the search index. The write performance of Tablestore is not affected when data is being asynchronously synchronized from a data table to a search index.

  In most cases, the latency generated when data is synchronized to a search index is within 3 seconds. You can view the latency in real time in the Tablestore console.

- Time to live (TTL)

  - If the UpdateRow operation is disabled for a data table, you can use the TTL feature of the search index that is created for the data table. For more information, see TTL of search indexes.

  - If you want to retain the data only for a period of time and the time field does not need to be updated, you can implement the TTL feature by splitting a data table into several time-specific data tables.

    This solution is implemented based on the following principles and rules, and has the following benefits:

    - Principle: Split a data table based on fixed periods of time, such as day, week, month, or year. Then, you can create a search index for each table. This way, tables for the specified periods of time are retained.

      For example, if you want to retain data for six months, you can store the data for each month in a data table (such as table_1, table_2, table_3, table_4, table_5, and table_6) and create a search index for each data table. Each data table and search index store the data only of a single month. Then, you only need to delete data tables that are retained for more than six months to implement the same feature as TTL.

      When you query data by using a search index, you only need to query one table if data that meets the time range requirement is in that table. If data that meets the time range requirement is included in multiple tables, you need to query all these tables and then combine the query results.

    - Rule: The size of a single table or search index cannot exceed 50 billion rows. The search index feature provides the optimal query performance if the size of a single table or search index does not exceed 20 billion rows.
    - Benefits:
      - You can adjust the data storage duration based on the number of data tables retained.
      - Query performance is directly proportional to data volumes. After a data table is split into multiple data tables, the data volume of each data table has an upper limit. This helps ensure better query performance and avoid high query latencies or timeouts.

- Max versions

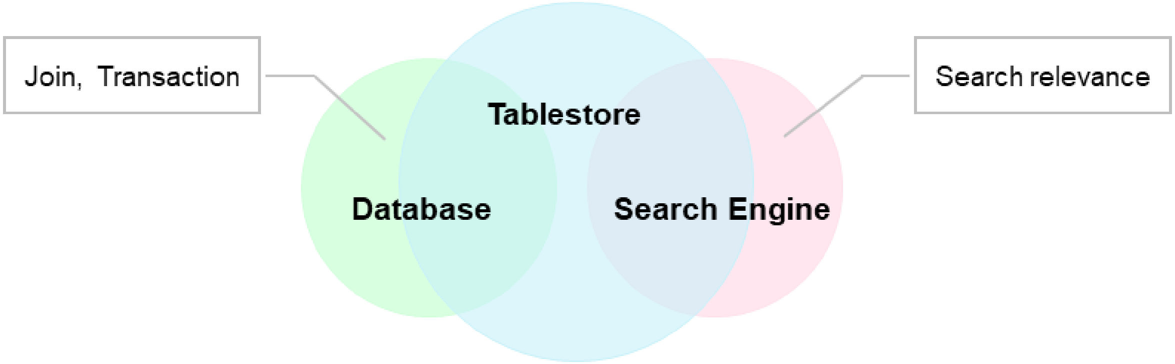
  You cannot create a search index for a data table for which you have specified the max versions parameter.

  You can customize the timestamp when you write data to a column that allows only a single version. If you first write data with a greater version number and then write data with a smaller version number, the data with the greater version number may be overwritten by data with the smaller version number.

Features

Search indexes can solve complex query problems in big data scenarios. Other systems such as databases and search engines can also solve data query problems. The following figure shows the differences between Tablestore, databases, and search engines.

Tablestore can provide all features of databases and search engines except for JOIN operations, transactions, and relevance of search results. Tablestore also has high data reliability of databases and supports advanced queries of search engines. Therefore, Tablestore can be used to replace the common architecture that consists of `databases and search engines`. If you do not need JOIN operations, transactions, and relevance of search results, we recommend that you use the search index feature of Tablestore.



Billing

For more information, see Billable items of search indexes.

6.2. Features

This topic describes the core features of search indexes and their equivalent SQL statements.

Core features

- Queries based on non-primary key columns

  Tablestore supports only queries that are based on primary key columns or prefixes of primary key columns. Therefore, Tablestore can be applied only to some query scenarios. You can use non-primary key columns to query data in a column for which you need to create a search index.

- Boolean query

  Boolean query is suitable for order scenarios. In order scenarios, a table may contain dozens of fields. It may be difficult to determine how to combine the fields required for queries when you create a table. Even if you have determined how to combine the required fields, hundreds of combinations may be available. If you use a relational database service, you have to create hundreds of indexes. In addition, if a combination is not specified in advance, you cannot query the required data.

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

- Queries by geographical location

  As mobile devices gain popularity, geographical location data becomes increasingly important. This data is used in most apps such as social media apps, food delivery apps, sports apps, and Internet of Vehicles (IoV) apps. These apps must support query features for the geographical location data that these apps provide.

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

  - Near: queries points within a specified radius based on a central point, such as the People Nearby feature in WeChat.
  - Within: queries points within a specified rectangular or polygonal area.

  Tablestore allows you to use these features to query geographical location data without using other database services or search engines.

- Full-text search

Search index can tokenize data to perform full-text search. However, unlike search engines, Tablestore returns only BM25-based results. Tablestore does not return custom relevant results in response to a query. If you need to query relevant results, we recommend that you use search engines.

Search index supports the following tokenization methods: single-word tokenization, delimiter tokenization, minimum semantic unit tokenization, maximum semantic unit tokenization, and fuzzy tokenization. For more information, see Tokenization.

- Fuzzy search

  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 queries by prefix. This feature is applicable to any natural languages. For example, in a query based on prefix "apple", words such as "apple6s" and "applexr" may be returned.

- Nested query

  In addition to a flat structure, online data such as labeled pictures has some complex and 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 weight score. The score is evaluated based on the size and position of an element in a picture. Therefore, each picture has multiple labels. Each label has a name and a weight score. You can use nested queries to query data based on the data labels.

  The following example provides an example of JSON data with labels:

```
{
 "tags": [
   {
      "name": "car",
      "score": 0.78
   },
   {
      "name": "tree",
      "score": 0.24
   }
 ]
}
```

  You can use nested queries to store and query multilayered data. This feature makes it easy to model complex data.

- Data deduplication

  Search index supports to filter out repeated data in query results. This feature 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 only products of a single brand, which is not user-friendly. You can use the data deduplication feature of Tablestore to resolve this issue.

- Sorting

In Tablestore, table data is sorted in alphabetical order of their primary key columns. To sort data based on other fields, you must use the sorting feature of search index. Tablestore supports a variety of sorting methods, including ascending, descending, single-field, and multi-field sorting. By default, returned results are sorted by the order of primary key values. Data in a search index is globally sorted.

- Total number of rows

  You can specify the number of rows that Tablestore returns for the current request when you use search index for a query. If you do not specify query conditions for the search index, Tablestore returns the total number of rows for which you have created indexes. After you stop writing new data to a table and create indexes for all attributes, Tablestore returns the total number of rows in the table. This feature applies to data verification and data-driven operations.

SQL

The following table lists the SQL statements and their equivalent search index.

SQL	Search index
Show	API operation: DescribeSearchIndex
Select	Parameter: ColumnsToGet
From	Parameter: index name Notice Supported for single-column indexes.
Where	Query: a variety of queries such as term query
Order by	Parameter: sort
Limit	Parameter: limit
Delete	API operations: query followed by DeleteRow
Like	Query: WildcardQuery
And	Parameter: operator = and
Or	Parameter: operator = or
Not	Query: BoolQuery(mustNotQueries)
Between	Query: RangeQuery
Null	Query: ExistsQuery
In	Query: TermsQuery
Min	Aggregation: min
Max	Aggregation: max

SQL	Search index
Avg	Aggregation: avg
Count	Aggregation: count
Count(distinct)	Aggregation: distinctCount
Sum	Aggregation: sum
Group By	GroupBy

6.3. Data type mappings

This topic describes the mappings between field data types in data tables and search indexes, and the additional attributes supported by different data types of fields.

The value of a field in a search index is the value of the field with the same name in the data table for which the search index is created. The data types of the two values must match. The following table describes the matching rules.

Notice The data types of the values in the search index must match the data types in the data table based on the rules described in the following table. Otherwise, Tablestore discards the data as dirty data. Make sure that field values of the Geo-point and Nested types comply with the formats described in the following table. If the formats do not match, Tablestore discards the data as dirty data. In this case, the data may be available in the data table but unavailable in the search index for queries.

Field data type in search indexes	Field data type in data tables	Description
Long	Integer	64-bit long integers.
Double	Double	64-bit double-precision 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 about tokenization, see Tokenization.
Date	Integer and String	The Date data type. You can specify the format of data of the Date type. For more information, see 日期数据类型.

Field data type in search indexes	Field data type in data tables	Description
Geo-point	String	The coordinate information of the location. This parameter value must be in the format of `latitude,longitude`. Valid values of the latitude: [-90,+90]. Valid values of the longitude: [-180,+180]. Example: `35.8,-45.91`.
Nested	String	The nested type. Example: [{"a": 1}, {"a": 3}].

The following table describes the additional attributes of fields in search indexes.

Attribute	Type	Description
Index	Boolean	Specifies whether to enable indexing. • The value of true indicates that Tablestore indexes the column with an inverted indexing schema or a spatio-temporal indexing schema. • The value of false indicates that Tablestore does not enable indexing for the column. If the column is not indexed, you cannot query data based on the column.
EnableSortAndAgg	Boolean	Specifies whether to enable the sorting and aggregation features. • The value of true indicates that the column can be used for sorting and aggregation. • The value of false indicates that the column cannot be used for sorting and aggregation.
Store	Boolean	Specifies whether to store the value of the column in the search index. The value of true indicates that Tablestore stores the value of the column in the search index. You can query the search index for the value of the column without the need to query the data table. This improves the query performance.
IsArray	Boolean	Specifies whether the value of the column is an array. The value of true indicates that the value of 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 specify the IsArray attribute for a nested column because a nested column is an array. Data of the ARRAY type can be used in all queries because arrays do not affect queries.

Attribute	Type	Description
isVirtualField	Boolean	Specifies whether the column is a virtual column. ● If you set this parameter to true, the column is a virtual column. If the column is a virtual column, you must specify the source field that is mapped to the column. For more information about virtual columns, see Virtual columns. ● If you set this parameter to false, the column is not a virtual column. In this case, the data type of the column in the search index must match the data type of the column in the data table.

The following table describes the combinations of data types and field attributes.

Type	Index	EnableSortAndAgg	Store	IsArray	isVirtualField
Long	Supported	Supported	Supported	Supported	Supported
Double	Supported	Supported	Supported	Supported	Supported
Boolean	Supported	Supported	Supported	Supported	Not supported
Keyword	Supported	Supported	Supported	Supported	Supported
Text	Supported	Not supported	Supported	Supported	Supported
Date	Supported	Supported	Supported	Supported	Supported
Geo-point	Supported	Supported	Supported	Supported	Supported
Nested	Applies only to child fields.	Applies only to child fields.	Applies only to child fields.	Nested fields are arrays.	Not supported

6.4. Quick start

6.4.1. Use the Tablestore console

Search indexes use inverted indexes and column stores to address complex query needs when a large amount of data exists. After you create a search index, you can use the search index to query data.

Prerequisites

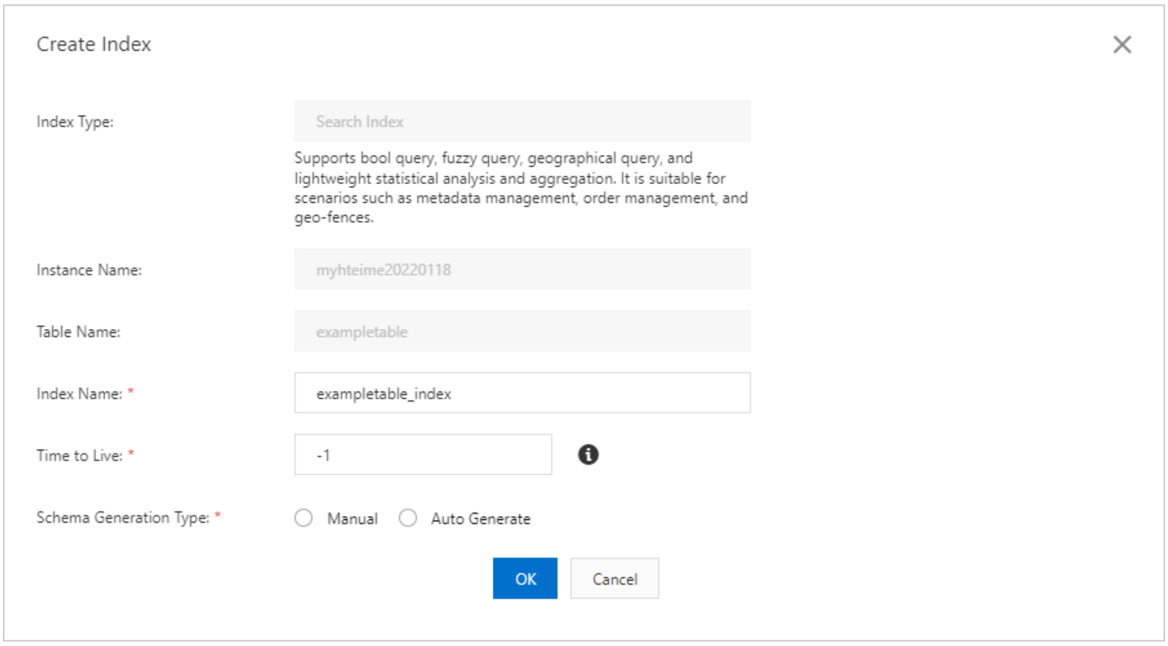
A data table for which the max versions parameter is set to 1 is created. For more information, see Create a data table.

Step 1: Create a search index

1. Log on to the Tablestore console.
2. On the **Overview** page, click the name of the desired instance or click Manage Instance in the

Actions column of the desired instance.

3. In the **Tables** section of the **Instance Details** tab, click the name of a data table and then click the **Indexes** tab. You can also click Indexes in the Actions column of the data table.
4. On the **Indexes** tab, click **Create Search Index**.
5. In the **Create Index** dialog box, configure the parameters that are required to create a search index.



Create Index

Index Type: Search Index

Supports bool query, fuzzy query, geographical query, and lightweight statistical analysis and aggregation. It is suitable for scenarios such as metadata management, order management, and geo-fences.

Instance Name: myhteime20220118

Table Name: exampletable

Index Name: * exampletable_index

Time to Live: * -1

Schema Generation Type: * ○ Manual ○ Auto Generate

   i. By default, the system generates a search index name. You can also set Index Name to a specific value.
   ii. Set Schema Generation Type.

> **Notice** The **Field Name** and **Field Type** values must match those in the data table. For more information about the mappings between fields in data tables and search indexes, see Data type mappings.

- If you set Schema Generation Type to **Manual**, set field names and field types for the field values. Specify whether to turn on Array.
- If you set Schema Generation Type to **Auto Generate**, the system automatically uses the primary key columns and attribute columns of the data table as indexed fields. Set field types for the field values and specify whether to turn on Array based on your business requirements.

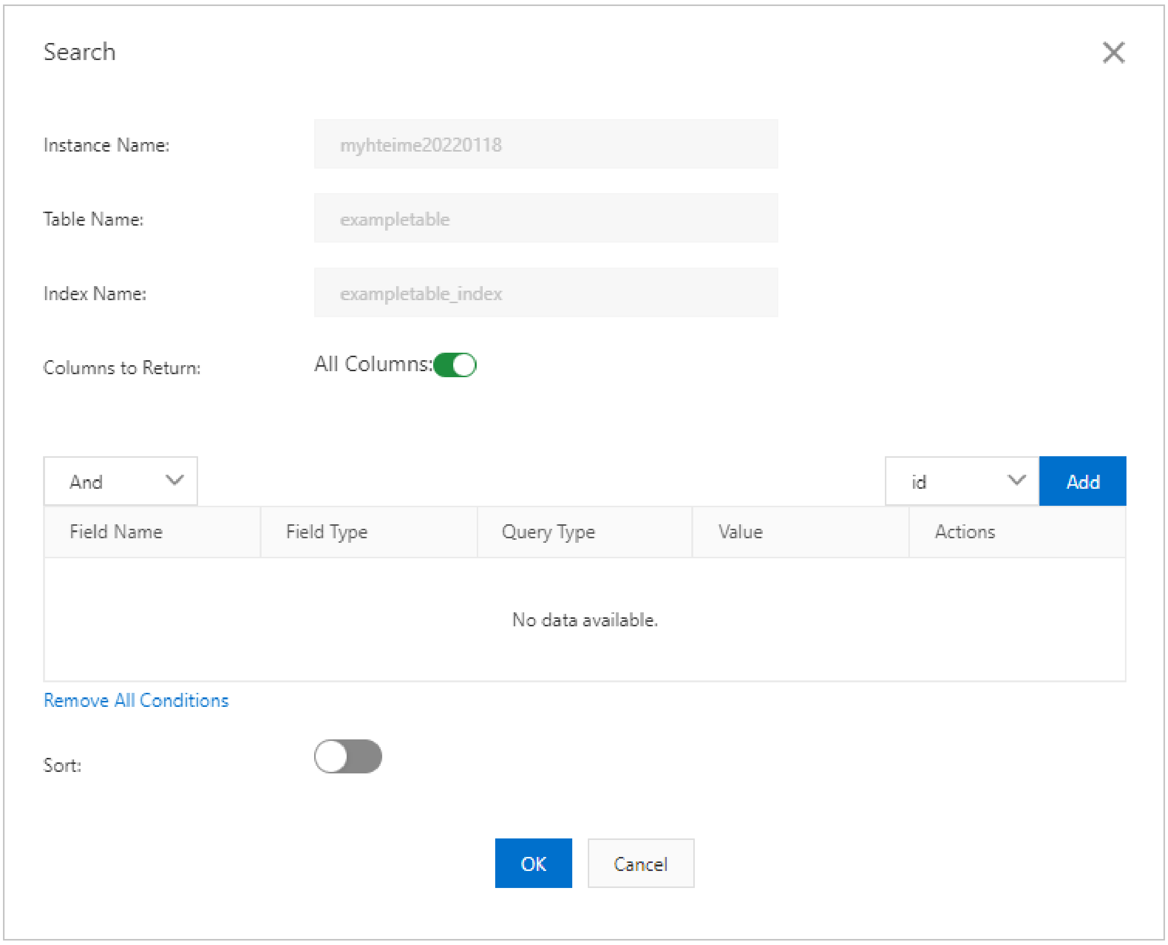
> **Note** To optimize performance in some cases, you can use virtual columns. For more information about virtual columns, see Virtual columns.

6. Click **OK**.

After the search index is created, click **Index Details** in the Actions column of the search index. You can view the information about the search index, such as the metering information and index fields.

Step 2: Query data

1. Log on to the Tablestore console.
2. On the **Overview** page, click the name of the required instance or click Manage Instance in the **Actions** column of the required instance.
3. In the **Tables** section of the **Instance Details** tab, click the name of a data table and then click the **Indexes** tab. You can also click Indexes in the Actions column of the data table.
4. On the **Indexes** tab, find the search index that you want to use to query data and click **Manage Data** in the Actions column.



Search

Instance Name: myhteime20220118

Table Name: exampletable

Index Name: exampletable_index

Columns to Return: All Columns: ●

Field Name	Field Type	Query Type	Value	Actions
No data available.				

Sort: ○

5. In the **Search** dialog box, configure query conditions.
   i. By default, the system returns all attribute columns. To return specified attribute columns, turn off **All Columns** and enter the attribute columns that you want to return. Separate multiple attribute columns with commas (,).

      > **Note** By default, the system returns all primary key columns of data tables.

   ii. Select indexed fields. Click **Add**. Set query methods and values for the fields.
   iii. By default, the sorting feature is disabled. To enable sorting, turn on **Sort** to sort query results based on the indexed fields. Add the indexed fields based on which the query results are sorted and configure sorting methods.
6. Click **OK**.

   Data that meets the query conditions is displayed in the specified order on the **Indexes** tab.

Related operations

- If you want to add, update, or delete indexed columns from a search index, you can use the feature that allows you to dynamically modify the schema of search indexes. For more information, see Dynamically modify schemas.
- If you want to query new fields or data of new field types without modifying the storage schema and the data in data tables, you can use the virtual column feature. For more information, see Virtual columns.

6.4.2. Use the Tablestore CLI

Prerequisites

- The Tablestore CLI is downloaded. For more information, see Download the Tablestore CLI.
- The instance is started and configured. For more information, see Start the Tablestore CLI and configure access information.
- An AccessKey pair is obtained. For more information, see Obtain an AccessKey pair.
- A data table for which the max versions parameter is set to 1 is created and used. For more information, see Create and use a data table.

Step 1: Create a search index

1. Run the **create_search_index** command to create a search index named search_index.

   ```
   create_search_index -n search_index
   ```

2. The following sample code shows how to enter the index schema as prompted:

   The index schema includes the settings of the search index (IndexSetting), the list of field schemas (FieldSchemas), and presorting settings for the search index (IndexSort). For more information about index schemas, see Create search indexes.

   ```
   {
       "IndexSetting": {
           "RoutingFields": null
       },
       "FieldSchemas": [
           {
               "FieldName": "gid",
               "FieldType": "LONG",
               "Index": true,
               "EnableSortAndAgg": true,
               "Store": true,
               "IsArray": false,
               "IsVirtualField": false
           },
           {
               "FieldName": "uid",
               "FieldType": "LONG",
               "Index": true,
               "EnableSortAndAgg": true,
               "Store": true,
               "IsArray": false,
               "IsVirtualField": false
   ```

```
            "IsVirtualField": false
        },
        {
            "FieldName": "col2",
            "FieldType": "LONG",
            "Index": true,
            "EnableSortAndAgg": true,
            "Store": true,
            "IsArray": false,
            "IsVirtualField": false
        },
        {
            "FieldName": "col3",
            "FieldType": "TEXT",
            "Index": true,
            "Analyzer": "single_word",
            "AnalyzerParameter": {
                "CaseSensitive": true,
                "DelimitWord": null
            },
            "EnableSortAndAgg": false,
            "Store": true,
            "IsArray": false,
            "IsVirtualField": false
        },
        {
            "FieldName": "col1",
            "FieldType": "KEYWORD",
            "Index": true,
            "EnableSortAndAgg": true,
            "Store": true,
            "IsArray": false,
            "IsVirtualField": false
        },
        {
            "FieldName": "col3V",
            "FieldType": "LONG",
            "Index": true,
            "EnableSortAndAgg": true,
            "Store": true,
            "IsArray": false,
            "IsVirtualField": true,
            "SourceFieldNames": [
                "col3"
            ]
        }
    ]
}
```

Step 2: Query data

1. Run the search command to use the search_index search index to query data and return all indexed columns of each row that meets the query conditions.

```
search -n search_index --return_all_indexed
```

2. The following sample code shows how to enter the query conditions as prompted by the system:

   Search indexes support query methods such as match all query (MatchAllQuery), match query (MatchQuery), match phrase query (MatchPhraseQuery), term query (TermQuery), terms query (TermsQuery), and prefix query (PrefixQuery). In this example, term query is used. For more information about term query, see Term query.

   ```
   {
       "Offset": -1,
       "Limit": 10,
       "Collapse": null,
       "Sort": null,
       "GetTotalCount": true,
       "Token": null,
       "Query": {
           "Name": "TermQuery",
           "Query": {
               "FieldName": "uid",
               "Term": 10001
           }
       },
       "Aggregations": [{
           "Name": "avg",
           "Aggregation": {
               "AggName": "agg1",
               "Field": "pid"
           }
       }]
   }
   ```

6.5. Use Tablestore SDKs

Prerequisites

- A data table for which the max versions parameter is set to 1 is created.
-

API operations

Category	Operation	Description
Management operations	CreateSearchIndex	Creates a search index.
	DescribeSearchIndex	Queries details about a search index.
	ListSearchIndex	Queries the list of search indexes.
	DeleteSearchIndex	Deletes a search index.

Category	Operation	Description
Query operations	Search	Implements all query features and analysis features such as sorting and aggregation. The results are returned in a specific order.
	ParallelScan	Implements all query features. You cannot call this operation to sort or aggregate data. Data that meets the query conditions is returned quickly. When you call this operation, you must call the ComputeSplits operation to query the maximum number of parallel scan tasks for a single ParallelScan request.

Procedure

1. Create a search index. For more information, see Create search indexes.
2. Call the Search or ParallelScan operation to query data. The following table describes the query methods that you can use to query data when you call the Search or ParallelScan operation.

Query method	Query	Description
Match all query	MatchAllQuery	This query is used to query the total number of rows in a table or randomly retrieve multiple rows from a table.
Match query	MatchQuery	This query uses approximate matches to retrieve query results. The keyword that you use for a query and the column values are tokenized based on the analyzer that you specified. Then, a match query is performed based on the tokens. The OR logical operator is used to relate tokens. If the number of tokens in a row that match the tokens in the tokenized keyword reaches the minimum value that you specified, the row meets the query conditions.
Match phrase query	MatchPhraseQuery	This query is similar to match query. A row meets the query conditions only when the order and position of the tokens in the row match the order and position of the tokens that are contained in the tokenized keyword.
Term query	TermQuery	This query uses full and exact matches to retrieve query results, which is similar to string matching. If a TEXT column is queried and one of the tokens in a row exactly matches the keyword, the row meets the query conditions.

Query method	Query	Description
Terms query	TermsQuery	This query is similar to term query, but you can specify multiple keywords at the same time. If one of the tokens in a row matches one of the keywords, the row meets the query conditions.
Prefix query	PrefixQuery	This query retrieves data that contains the specified prefix. If a TEXT column is queried and one of the tokens in a row contains the specified prefix, the row meets the query conditions.
Range query	RangeQuery	This query retrieves data within a specified range. If a TEXT column is queried and one of the tokens in a row is within the specified range, the row meets the query conditions.
Wildcard query	WildcardQuery	This query retrieves data that matches a string that contains one or more wildcard characters. You can use the asterisk (*) and question mark (?) wildcard characters in a string. The asterisk (*) matches a string of any length in, before, or after a search term. The question mark (?) matches a single character in a specific position.
Boolean query	BoolQuery	You can use Boolean query to query rows based on a subquery or a combination of subqueries. Tablestore returns the rows that match the subquery or the combination of subqueries. Subquery conditions can be combined by using logical operators, such as AND, NOT, and OR.
Nested query	NestedQuery	You can use nested query to query the data of nested fields.
Geo-distance query	GeoDistanceQuery	You can specify a circular geographical area that consists of a central point and radius as a query condition. Tablestore returns the rows in which the value of the specified column falls within the geographical circular area.
Geo-bounding box query	GeoBoundingBoxQuery	You can specify a rectangular geographical area as a query condition. Tablestore returns the rows in which the value of the specified column falls within the rectangular geographical area.
Geo-polygon query	GeoPolygonQuery	You can specify a polygonal geographical area as a query condition. Tablestore returns the rows in which the value of the specified column falls within the polygonal geographical area.

Query method	Query	Description
Exists query	ExistQuery	Exists query is also called NULL query or NULL-value query. This query is used for sparse data to determine whether a column of a row exists. For example, you can query the rows in which the value of the address column is not empty. If a column does not exist in a row or the value of the column is an empty array ("[]"), the column does not exist in the row.

Related operations

- If you want to analyze data in a table, you can call the Search operation to use the aggregate feature. You can use the aggregate feature to query the maximum value, the sum of the values, and the number of rows. For more information, see Aggregation.
- If you do not need to sort the rows that meet the query conditions and you want to quickly obtain all rows that meet the query conditions, you can call the ParallelScan and ComputeSplits operations to use the parallel scan feature. For more information, see Parallel scan.
- When you call the Search operation to query data, you can sort or paginate the rows that meet the query conditions. For more information, see Sorting and pagination.
- When you call the Search operation to query data, you can use the collapse (distinct) feature to collapse the result set based on a specified column. This way, data of the specified type appears only once in the query results. For more information, see Collapse (distinct).
- If you want to perform full-text search, you must tokenize the field for which tokenization can be performed and select a suitable query method to query data. For more information, see Tokenization.
- If you want to store and query data in multiple logical relationships, you can store the data as a nested field and use nested query to query the data. For more information, see Nested and Nested query.
- If you want the system to automatically delete the data that is retained in a search index for a period of time that exceeds the specified duration, you can use the time to live (TTL) feature of the search index. For more information, see TTL of search indexes.
-
-

6.6. Basic features

6.6.1. Create search indexes

After you create a search index for a data table, you can query data in the data table based on the fields for which indexing is enabled in the search index. You can create multiple search indexes for a data table.

Optimal method to create search indexes

We recommend that you determine the number of search indexes that you want to create for a data table based on your query requirements.

If you have a data table that contains the id, name, age, city, and sex fields, you can use one of the following methods to create search indexes if you want to query data by name, age, or city:

- Method 1: Create a search index for each field

  If you use this method, you must create the following search indexes: name_index, age_index, and city_index.

  - To query students by city, use city_index. To query students by age, use age_index.
  - However, this method does not work if you want to query students who are younger than 12 years old and live in City A.

  The implementation of the method is similar to that of the global secondary index feature. However, this method is not cost-effective. We recommend that you use Method 2 to create the search index.

- Method 2: Create one search index for multiple fields

  In this method, a search index named student_index is created. The search index contains the following fields: name, age, and city.

  - To query students by city, query the city field in student_index. To query students by age, query the age field in student_index.
  - To query students who are younger than 12 years old and live in City A, query the age and city fields in student_index.

  This method makes full use of the advantages of search indexes and is more cost-effective. We recommend that you use this method to create a search index.

Limits

- Timeliness of search index creation

  After a search index is created, it takes a few seconds before you can use the search index. During this period, you can only write data to the data table, but you cannot query the metadata of the index or query data by using the index.

- Quantity

  For more information, see Search index limits.

API operation

You can call the CreateSearchIndex operation to create a search index.

Usage

You can use the following SDKs to create search indexes:

- Tablestore SDK for Java: Create search indexes
- Tablestore SDK for Go: Create search indexes
- Tablestore SDK for Python: Create search indexes
- Tablestore SDK for Node.js: Create search indexes
- Tablestore SDK for .NET: Create search indexes
- Tablestore SDK for PHP: Create search indexes

Parameters

When you create a search index, you must specify tableName, indexName, and indexSchema. In indexSchema, you must specify fieldSchemas, indexSetting, and indexSort. The following table describes the parameters.

Parameter	Description
tableName	The name of the data table.
indexName	The name of the search index.

Parameter	Description
fieldSchemas	The list of field schemas. Each field schema contains the following parameters: • fieldName: This parameter is required and specifies the name of the field in the search index. The value is used as a column name. Type: String. A field in a search index can be a primary key column or an attribute column. • fieldType: This parameter is required and specifies the type of the field. Use FieldType.XXX to set the type. For more information, see Data types of column values. • array: This parameter is optional and specifies whether the value is an array. Type: Boolean. If you set this parameter to true, the column stores data as an array. Data written to the column must be a JSON array. Example: ["a","b","c"]. The values of fields of the Nested type are arrays. If you set fieldType to Nested, skip this parameter. • index: This parameter is optional and specifies whether to enable indexing for the column. Type: Boolean. Default value: true. A value of true indicates that Tablestore indexes the column with an inverted indexing schema or a spatio-temporal indexing schema. A value of false indicates that Tablestore does not enable indexing for the column. • analyzer: This parameter is optional and specifies the type of the analyzer that you want to use. If fieldType is set to Text, you can configure this parameter. Otherwise, the default analyzer type single-word tokenization is used. For more information about tokenization, see Tokenization. • enableSortAndAgg: This parameter is optional and specifies whether to enable sorting and aggregation. Type: Boolean. Sorting can be enabled only for fields for which enableSortAndAgg is set to true. For more information about sorting, see Sorting and pagination. **Notice** Fields of the Nested type do not support sorting and aggregation, but subcolumns of fields of the Nested type support sorting and aggregation. • store: This parameter is optional and specifies whether to store the value of the field in the search index. Type: Boolean. If you set store to true, you can read the value of the field from the search index without querying the data table. This improves query performance. • isVirtualField: This parameter is optional and specifies whether the field is a virtual column. Type: Boolean. Default value: false. This parameter is required only when you use virtual columns. For more information about virtual columns, see Virtual columns. • sourceFieldName: This parameter is optional and specifies the name of the source field to which the virtual column is mapped in the data table. Type: String. This parameter is required when isVirtualField is set to true.

Parameter	Description
indexSetting	The settings of the search index, including routingFields. routingFields: This parameter is optional and specifies custom routing fields. You can specify some primary key columns as routing fields. Tablestore distributes data that is written to a search index across different partitions based on the specified routing fields. The data whose routing field values are the same is distributed to the same partition.
indexSort	The presorting settings of the search index, including sorters. If no value is specified for the indexSort parameter, field values are sorted by primary key by default. Note You can skip the presorting settings for search indexes that contain fields of the Nested type. sorters: This parameter is required and specifies the presorting method for the search index. PrimaryKeySort and FieldSort are supported. For more information, see Sorting and pagination. • PrimaryKeySort: Data is sorted by primary key. You can configure the following parameter for PrimaryKeySort: order: the sort order. Data can be sorted in ascending or descending order. Default value: SortOrder.ASC. • FieldSort: Data is sorted by field value. You can configure the following parameters for FieldSort: Only fields for which indexing is enabled and enableSortAndAgg is set to true can be presorted. ○ fieldName: the name of the field that is used to sort data. ○ order: the sort order. Data can be sorted in ascending or descending order. Default value: SortOrder.ASC. ○ mode: the sorting method that is used when the field contains multiple values.
timeToLive	This parameter is optional and specifies the retention period of data in the search index. Unit: seconds. Default value: -1. When the retention period exceeds the timeToLive value, Tablestore automatically deletes expired data. The minimum timeToLive value is 86400, which is equal to one day. A value of -1 specifies that data never expires. For more information about how to manage the time to live (TTL) of search indexes, see TTL of search indexes.

Examples

- Create a search index

  The following sample code shows how to create a search index that consists of the Col_Keyword and Col_Long columns. Set the type of data in Col_Keyword to String and Col_Long to Long.

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); // Specify the name of the data table.
    request.setIndexName(indexName); // Specify the name of the search index.
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD) // Specify the name and type of the field.
                    .setIndex(true) // Enable indexing.
                    .setEnableSortAndAgg(true) // Enable sorting and aggregation.
                    .setStore(true), // Specify that the value of the field is stored in the search index.
            new FieldSchema("Col_Long", FieldType.LONG)
                    .setIndex(true)
                    .setEnableSortAndAgg(true)
                    .setStore(true)));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); // Call a client to create the search index.
}
```

- Create a search index with indexSort specified

```
private static void createSearchIndexWithIndexSort(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName);
    request.setIndexName(indexName);
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD).setIndex(true).setEnableSortAndAgg(true).setStore(true),
            new FieldSchema("Col_Long", FieldType.LONG).setIndex(true).setEnableSortAndAgg(true).setStore(true),
            new FieldSchema("Col_Text", FieldType.TEXT).setIndex(true).setStore(true),
            new FieldSchema("Timestamp", FieldType.LONG).setIndex(true).setEnableSortAndAgg(true).setStore(true)));
    // Presort data by the Timestamp column. You must enable indexing and set enableSortAndAgg to true for the Timestamp column.
    indexSchema.setIndexSort(new Sort(
            Arrays.<Sort.Sorter>asList(new FieldSort("Timestamp", SortOrder.ASC))));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request);
}
```

- Create a search index with the TTL specified

Notice Make sure that updates to the data table are prohibited.

```
// Use Tablestore SDK for Java V5.12.0 or later to create a search index.
public void createIndexWithTTL(SyncClient client) {
    int days = 7;
    CreateSearchIndexRequest createRequest = new CreateSearchIndexRequest();
    createRequest.setTableName(tableName);
    createRequest.setIndexName(indexName);
    createRequest.setIndexSchema(indexSchema);
    // Specify the TTL for the search index.
    createRequest.setTimeToLiveInDays(days);
    client.createSearchIndex(createRequest);
}
```

- Create a search index with virtual columns specified

  The following sample code shows how to create a search index that contains columns Col_Keyword and Col_Long. Each of the columns has a virtual column. The virtual column of the Col_Keyword column is Col_Keyword_Virtual_Long and that of the Col_Long column is Col_Long_Virtual_Keyword. The Col_Keyword_Virtual_Long column is mapped to the Col_Keyword column in the data table, and the Col_Long_Virtual_Keyword column is mapped to the Col_Long column in the data table.

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); // Specify the name of the data table.
    request.setIndexName(indexName); // Specify the name of the search index.
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
        new FieldSchema("Col_Keyword", FieldType.KEYWORD) // Specify the name and type of
the field.
            .setIndex(true) // Enable indexing.
            .setEnableSortAndAgg(true) // Enable sorting and aggregation.
            .setStore(true),
        new FieldSchema("Col_Keyword_Virtual_Long", FieldType.LONG) // Specify the name a
nd type of the field.
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true) // Specify whether the field is a virtual column.
            .setSourceFieldName("Col_Keyword"), // Specify name of the source field to wh
ich the virtual column is mapped in the data table.
        new FieldSchema("Col_Long", FieldType.LONG)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true),
        new FieldSchema("Col_Long_Virtual_Keyword", FieldType.KEYWORD)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true)
            .setSourceFieldName("Col_Long")));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); // Call a client to create the search index.
}
```

6.6.2. TTL of search indexes

Time to live (TTL) is an attribute of search indexes that specifies the retention period of data in search indexes. When data in a search index is retained for a period of time that exceeds the TTL value, Tablestore automatically deletes the data from the search index to free up storage space and reduce costs.

Usage notes

- To use the TTL feature of a search index, you must prohibit the UpdateRow operation on the data table for which the search index is created due to the following reasons:

  The TTL feature of data tables takes effect on attribute columns, and the TTL feature of search indexes takes effect on the entire rows. If the UpdateRow operation is performed on a data table, when the system clears data in the data table, the values of some fields are deleted and the values of some fields are retained in the data table. However, the entire rows in the search index that is created for the data table are not deleted. As a result, data in the data table and search index is inconsistent.

  If the UpdateRow operation is required, check whether the UpdateRow operation can be changed to the PutRow operation.
- The TTL value of search indexes can be -1 or a positive int32 in seconds. The value of -1 indicates that data in the search index never expires and the maximum int32 value is equivalent to approximately 68 years.
- The TTL value of a search index is independent of and must be smaller than or equal to the TTL value of the data table for which the search index is created. If you need to change TTL values of search indexes and data tables for which the search indexes are created to smaller values, you must change the TTL values of the search indexes before you change the TTL values of the data tables.
- Tablestore automatically deletes expired data from search indexes every day. In some cases, you can still query expired data in search indexes. Tablestore automatically deletes the expired data in the next cycle.
- After you change the TTL values of data tables and search indexes, the system automatically deletes legacy expired data from the data tables and search indexes in the next cycle.

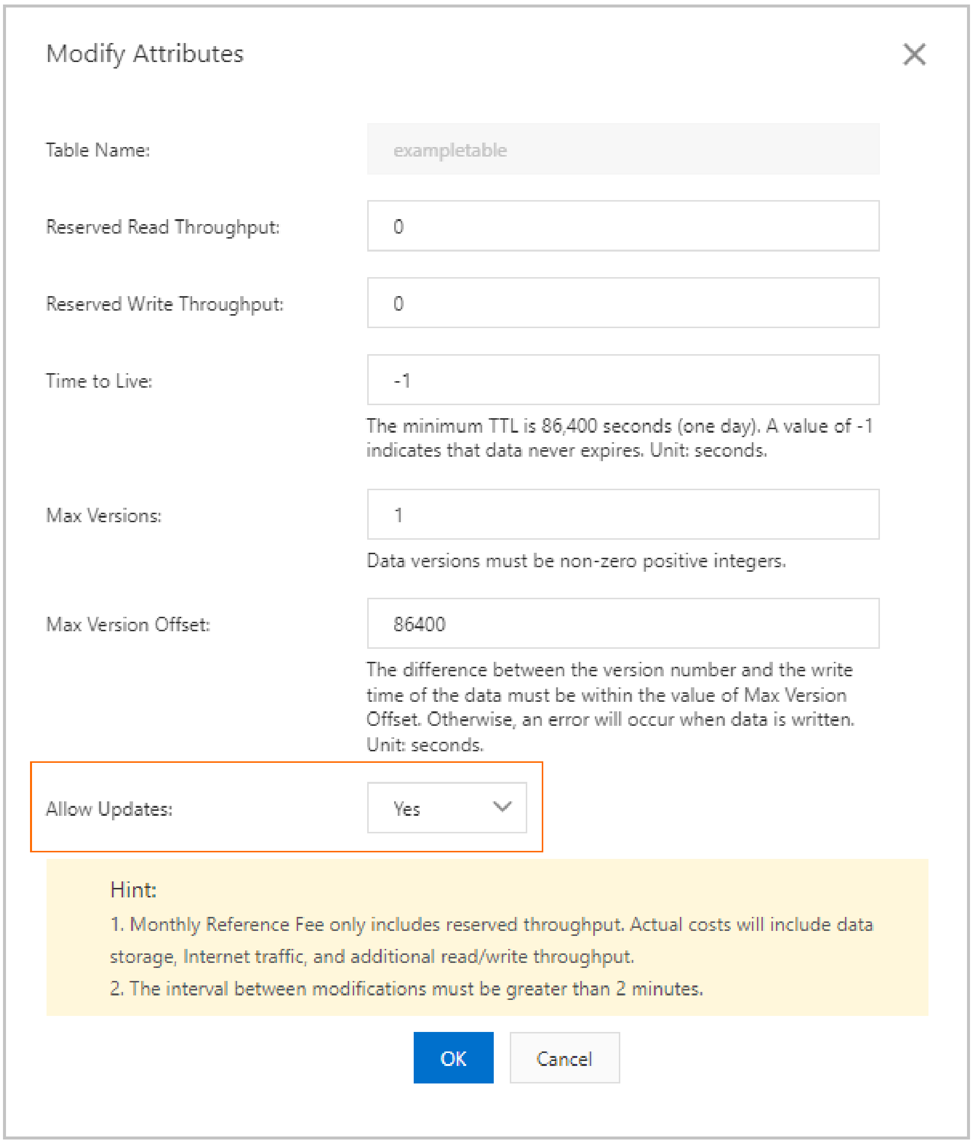
Procedure

You can configure the TTL of a search index by using the Tablestore console or Tablestore SDKs. To use the TTL feature of a search index, you must prohibit the UpdateRow operation on the data table for which the search index is created.

Use the Tablestore console

1. Prohibit the UpdateRow operation on a data table.
    i. On the **Basic information** tab of the data table, click **Modify Attributes**.

ii. In the **Modify Attributes** dialog box, select **No** for Allow Updates and click **OK**.



Modify Attributes

Table Name: exampletable

Reserved Read Throughput: 0

Reserved Write Throughput: 0

Time to Live: -1

The minimum TTL is 86,400 seconds (one day). A value of -1 indicates that data never expires. Unit: seconds.

Max Versions: 1

Data versions must be non-zero positive integers.

Max Version Offset: 86400

The difference between the version number and the write time of the data must be within the value of Max Version Offset. Otherwise, an error will occur when data is written. Unit: seconds.

Allow Updates: Yes

Hint:

1. Monthly Reference Fee only includes reserved throughput. Actual costs will include data storage, Internet traffic, and additional read/write throughput.

2. The interval between modifications must be greater than 2 minutes.

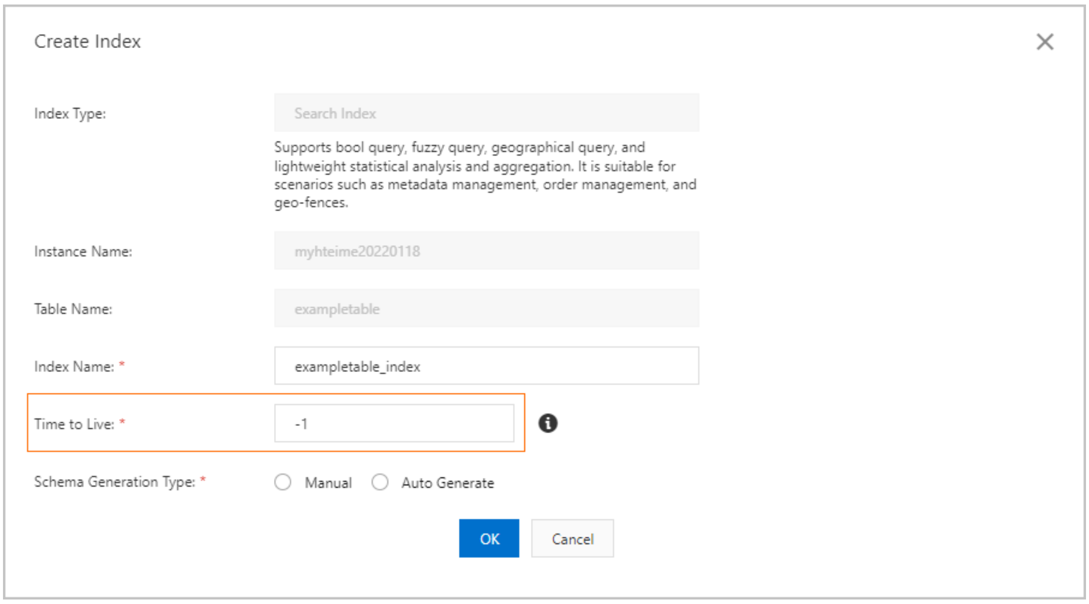
2. Specify the TTL for a search index.

   After the UpdateRow operation on a data table is prohibited, you can specify the TTL of a search index when you create the search index or modify the TTL of existing search indexes.

   - Specify the TTL when you create a search index

     a. On the **Indexes** tab of the data table for which you want to create a search index and on which the UpdateRow operation is prohibited, click **Create Search Index**.

b. In the **Create Index** dialog box, configure the Index Name, Time to Live, and Schema Generation Type parameters, and click **OK**.



Create Index

Index Type: Search Index

Supports bool query, fuzzy query, geographical query, and lightweight statistical analysis and aggregation. It is suitable for scenarios such as metadata management, order management, and geo-fences.

Instance Name: myhteime20220118

Table Name: exampletable

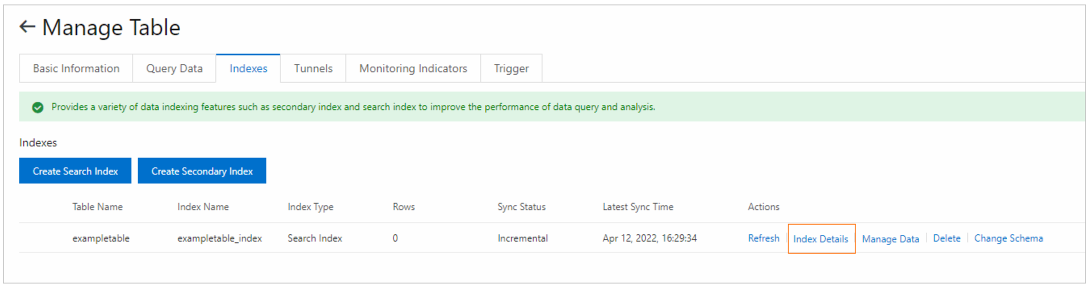
Index Name: * exampletable_index

Time to Live: * -1

Schema Generation Type: * ○ Manual ○ Auto Generate

- Modify the TTL for an existing search index

    a. On the **Indexes** tab of the data table on which the UpdateRow operation is prohibited, find the search index for which you want to modify the TTL and click **Index Details** in the Actions column.



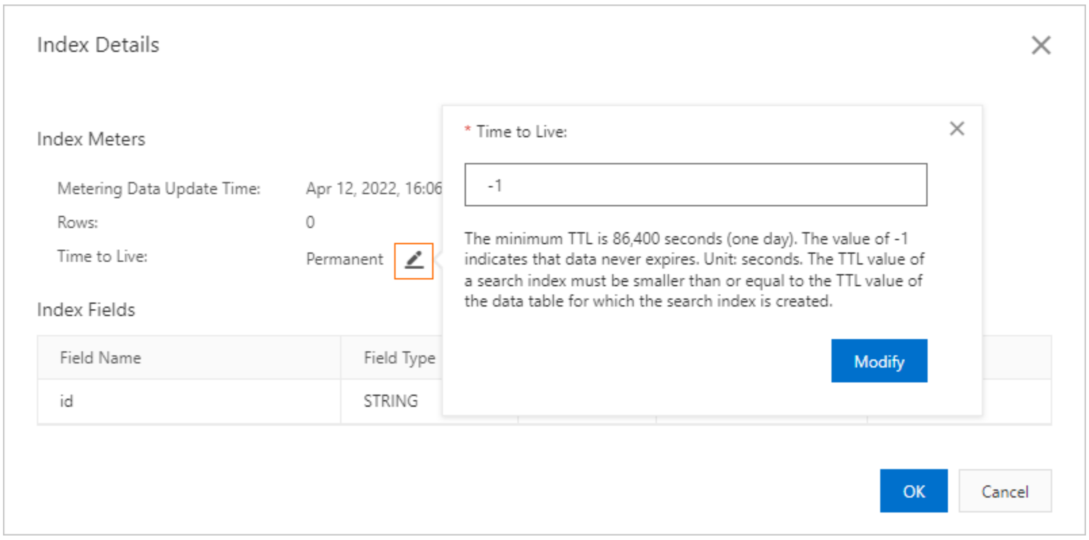
← Manage Table

Provides a variety of data indexing features such as secondary index and search index to improve the performance of data query and analysis.

Indexes

Table Name	Index Name	Index Type	Rows	Sync Status	Latest Sync Time	Actions
exampletable	exampletable_index	Search Index	0	Incremental	Apr 12, 2022, 16:29:34	Refresh | Index Details | Manage Data | Delete | Change Schema

    b. In the **Index Details** dialog box, click the ✎ icon, modify the value of the Time to Live parameter, and then click **Modify**.



Index Details

Index Meters

Metering Data Update Time: Apr 12, 2022, 16:06

Rows: 0

Time to Live: Permanent

Index Fields

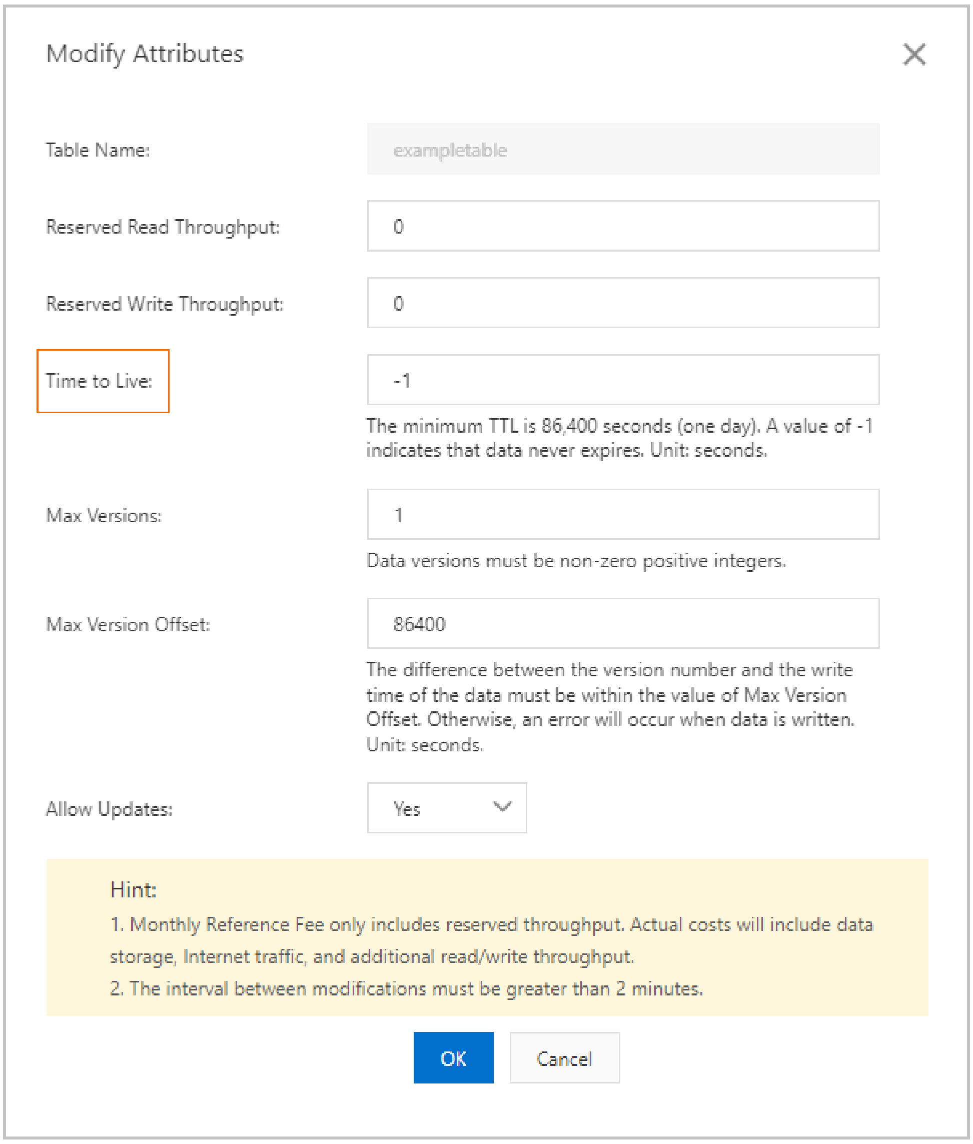
Field Name	Field Type
id	STRING

* Time to Live: -1

The minimum TTL is 86,400 seconds (one day). The value of -1 indicates that data never expires. Unit: seconds. The TTL value of a search index must be smaller than or equal to the TTL value of the data table for which the search index is created.

c. Click **OK**.

3. The TTL of a data table is independent of the TTL of the search index that is created for the data table. If you want to use the TTL of a data table, configure the TTL for the data table.
   i. In the **Description** section of the **Basic Information** tab, click **Modify Attributes**.
   ii. In the **Modify Attributes** dialog box, configure the Time to Live parameter based on your business requirements, and click **OK**.



Modify Attributes

Table Name:	exampletable
Reserved Read Throughput:	0
Reserved Write Throughput:	0
Time to Live:	-1 The minimum TTL is 86,400 seconds (one day). A value of -1 indicates that data never expires. Unit: seconds.
Max Versions:	1 Data versions must be non-zero positive integers.
Max Version Offset:	86400 The difference between the version number and the write time of the data must be within the value of Max Version Offset. Otherwise, an error will occur when data is written. Unit: seconds.
Allow Updates:	Yes

Hint:
1. Monthly Reference Fee only includes reserved throughput. Actual costs will include data storage, Internet traffic, and additional read/write throughput.
2. The interval between modifications must be greater than 2 minutes.

Use Tablestore SDKs

1. Prohibit the UpdateRow operation on a data table.

```
public void disableTableUpdate(SyncClient client) {
    UpdateTableRequest updateTableRequest = new UpdateTableRequest(tableName);
    TableOptions options = new TableOptions();
    // Prohibit the UpdateRow operation on a data table to prevent impacts on your busi
ness.
    options.setAllowUpdate(false);
    updateTableRequest.setTableOptionsForUpdate(options);
    client.updateTable(updateTableRequest);
}
```

2. Specify the TTL for search indexes.

   After the UpdateRow operation on a data table is prohibited, you can specify the TTL of a search index when you create the search index or modify the TTL of existing search indexes.

   - Specify the TTL when you create a search index

     ```
     ```

   - Modify the TTL for an existing search index

     ```
     // Use Tablestore SDK for Java V5.12.0 or later.
     public void updateIndexWithTTL(SyncClient client) {
         int days = 7;
         UpdateSearchIndexRequest updateSearchIndexRequest = new UpdateSearchIndexRequest(
     tableName, indexName);
         // Modify the TTL for the search index.
         updateSearchIndexRequest.setTimeToLiveInDays(days);
         client.updateSearchIndex(updateSearchIndexRequest);
     }
     ```

3. The TTL of a data table is independent of the TTL of the search index that is created for the data table. If you want to use the TTL of a data table, configure the TTL for the data table.

   ```
   public void updateTableTTL(SyncClient client) {
       int days = 7;
       UpdateTableRequest updateTableRequest = new UpdateTableRequest(tableName);
       TableOptions options = new TableOptions();
       options.setTimeToLiveInDays(days);
       updateTableRequest.setTableOptionsForUpdate(options);
       client.updateTable(updateTableRequest);
   }
   ```

6.6.3. Query the description of a search index

This topic describes how to query the description of a search index, including the information of columns in the search index and configurations of the search index.

Operations

You can call the DescribeSearchIndex operation to query the description of a search index.

Use Tablestore SDKs

You can use the following Tablestore SDKs to query the description of a search index:

- Tablestore SDK for Java: Query the description of a search index

- Tablestore SDK for GO: Query the description of a search index
- Tablestore SDK for Python: Query the description of a search index
- Tablestore SDK for Node.js: Query the description of a search index
- Tablestore SDK for .NET SDK: Query the description of a search index
- Tablestore SDK for PHP SDK: Query the description of a search index

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.

Examples

```
private static DescribeSearchIndexResponse describeSearchIndex(SyncClient client) {
    DescribeSearchIndexRequest request = new DescribeSearchIndexRequest();
    request.setTableName(TABLE_NAME); request.setTableName(TABLE_NAME); // Set the name of
the table.
    request.setIndexName(INDEX_NAME); // Set the name of the search index.
    DescribeSearchIndexResponse response = client.describeSearchIndex(request);
    System.out.println(response.jsonize()); // Display the details of the response.
    System.out.println(response.getSyncStat().getSyncPhase().name());// Display the synchro
nization status of data in the search index.
    return response;
}
```

6.6.4. ARRAY and Nested field types

Search index provides the following special field types in addition to the basic field types such as LONG, DOUBLE, BOOLEAN, KEYWORD, TEXT, and GEOPOINT:

ARRAY

ARRAY is a type that can be combined with basic field types such as LONG, DOUBLE, BOOLEAN, KEYWORD, TEXT, and GEOPOINT. For example, the combination of LONG with ARRAY is used to specify arrays of the LONG INTEGER type. LONG ARRAY fields can contain multiple long integers. If a query matches a component of an array, the corresponding row is returned.

The following table describes arrays combined with basic field types.

ARRAY	Description
Long Array	An array of long integers. Example: [1000, 4, 5555].
Boolean Array	An array of BOOLEAN values. Example: [true, false].
Double Array	An array of double-precision floating-point numbers. Example: [3.1415926, 0.99].

ARRAY	Description
Keyword Array	An array of strings in the JSON ARRAY format. Example: [\"Hangzhou\", \"Xi'an\"].
Text Array	An array of text in the JSON ARRAY format. Example: [\"Hangzhou\", \"Xi'an\"]. TEXT JSON arrays are not commonly used.
GeoPoint Array	An array of latitude and longitude coordinate pairs. Example: [\"34.2, 43.0\", \"21.4, 45.2\"].

The ARRAY type is supported only in search indexes. Therefore, when the type of an index field involves ARRAY, the field in the table must be of the STRING type. A basic data type such as LONG or DOUBLE in the search index remains the same as that in the table. If a price field is of the DOUBLE ARRAY type, the field type in the table must be STRING, and the field type must be DOUBLE and isArray must be set to true in the search index.

Nested

Nested indicates nested documents. Nested documents are used when a row of data (document) contains multiple child rows (child documents). Multiple child documents are stored in a Nested field. You must specify the schema of child rows in the Nested field. The schema must include the fields of the child rows and the property of each field. Nested is similar to ARRAY. However, Nested supports more features.

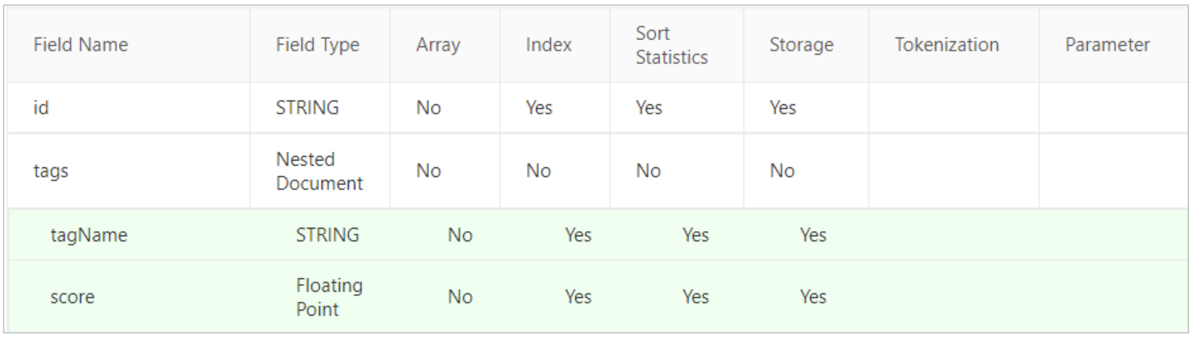
Nested fields are written as strings in JSON arrays to tables.

> **Notice** Even when a field contains a single child row, the written strings must be JSON arrays.

- Example of single-level Nested fields

  You can create single-level Nested fields in the console or by using Tablestore SDKs.

  This section provides an example on how to create a single-level Nested field by using Tablestore SDK for Java. A Nested field named tags is used in this example. Each child row contains two fields, as shown in the following figure.

Field Name	Field Type	Array	Index	Sort Statistics	Storage	Tokenization	Parameter
id	STRING	No	Yes	Yes	Yes		
tags	Nested Document	No	No	No	No		
tagName	STRING	No	Yes	Yes	Yes		
score	Floating Point	No	Yes	Yes	Yes		

  - Field name: tagName. Field type: KEYWORD.
  - Field name: score. Field type: DOUBLE.

  The following data samples are written to the table: `[{"tagName":"tag1", "score":0.8}, {"tagName":"tag2", "score":0.2}]` .

```
// Specify the FieldSchema class for the child rows.
List<FieldSchema> subFieldSchemas = new ArrayList<FieldSchema>();
subFieldSchemas.add(new FieldSchema("tagName", FieldType.KEYWORD)
    .setIndex(true).setEnableSortAndAgg(true));
subFieldSchemas.add(new FieldSchema("score", FieldType.DOUBLE)
    .setIndex(true).setEnableSortAndAgg(true));
// Specify that FieldSchema of the child rows is used as subfieldSchemas of the Nested field.
FieldSchema nestedFieldSchema = new FieldSchema("tags", FieldType.NESTED)
    .setSubFieldSchemas(subFieldSchemas);
```

- Example of multiple-level Nested fields

  You can create multiple-level Nested fields only by using Tablestore SDKs.

  This section provides an example on how to create a multiple-level Nested field by using Tablestore SDK for Java. A Nested field named user is used in this example. Each child row contains three fields of different basic field types and one Nested field.

  - Field name: name. Field type: KEYWORD.
  - Field name: age. Field type: LONG.
  - Field name: phone. Field type: KEYWORD.
  - Nested field name: address. Names of the fields contained in each child row: province, city, and street. All fields contained in each child row are of the KEYWORD type.

  The following data sample is written to the table: `[ {"name":"Zhangsan","age":20,"phone":"13900006666","address":[{"province":"Zhejiang Province","city":"Hangzhou City","street":"No. 1201, Xingfu Community, Yangguang Avenue"}]}]`

```
// Construct FieldSchema for each child row of the address Nested field. Each child row contains three fields. The path specified by user.address can be used to query data of fields in a child row.
List<FieldSchema> addressSubFiledSchemas = new ArrayList<>();
addressSubFiledSchemas.add(new FieldSchema("province",FieldType.KEYWORD));
addressSubFiledSchemas.add(new FieldSchema("city",FieldType.KEYWORD));
addressSubFiledSchemas.add(new FieldSchema("street",FieldType.KEYWORD));
// Construct FieldSchema for each child row of the user Nested field. Each child row contains three fields of different basic field types and one Nested field named address. The path specified by user can be used to query data of fields in a child row.
List<FieldSchema> subFieldSchemas = new ArrayList<>();
subFieldSchemas.add(new FieldSchema("name",FieldType.KEYWORD));
subFieldSchemas.add(new FieldSchema("age",FieldType.LONG));
subFieldSchemas.add(new FieldSchema("phone",FieldType.KEYWORD));
subFieldSchemas.add(new FieldSchema("address",FieldType.NESTED).setSubFieldSchemas(addressSubFiledSchemas));
// Specify that FieldSchema of the child rows in the user Nested field is used as subfieldSchemas of the Nested field.
List<FieldSchema> fieldSchemas = new ArrayList<>();
fieldSchemas.add(new FieldSchema("user",FieldType.NESTED).setSubFieldSchemas(subFieldSchemas));
```

The Nested type has the following limits:

- Nested indexes do not support the IndexSort feature. However, IndexSort can improve query performance in multiple scenarios.

- If you use a search index that contains a nested field to query data and require pagination, you must specify the sorting method to return data in the query conditions. Otherwise, Tablestore does not return nextToken when only part of data that meets the query conditions is read.
- Nested queries provide lower performance than other types of queries.

Apart from the preceding limits, the Nested type supports all queries, sorting, and aggregations.

6.6.5. List search indexes

After you create a search index, you can query the list of all search indexes created in the current instance or associated with a table.

Operations

You can call the ListSearchIndex operation to list search indexes.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement search index:

- Tablestore SDK for Java: List search indexes
- Tablestore SDK for Go: List search indexes
- Tablestore SDK for Python: List search indexes
- Tablestore SDK for Node.js: List search indexes
- Tablestore SDK for .NET: List search indexes
- Tablestore SDK for PHP: List search indexes

Parameters

Parameter	Description
tableName	Optional. This parameter specifies the name of the table. • If the name of the table is set, all search indexes associated with the table are returned. • If the name of the table is not set, all search indexes in the current instance are returned.

Examples

```
private static List<SearchIndexInfo> listSearchIndex(SyncClient client) {
    ListSearchIndexRequest request = new ListSearchIndexRequest();
    request.setTableName(TABLE_NAME); // Set the name of the table.
    return client.listSearchIndex(request).getIndexInfos(); // Return all search indexes as
sociated with the table.
}
```

6.6.6. Delete search indexes

This topic describes how to delete a search index created for a table.

Operations

You can call the DeleteSearchIndex operation to delete a search index.

Use Tablestore SDKs

You can use the following Tablestore SDKs to delete a search index:

- Tablestore SDK for Java: Delete search indexes
- Tablestore SDK for Go: Delete search indexes
- Tablestore SDK for Python: Delete search indexes
- Tablestore SDK for Node.js: Delete search indexes
- Tablestore SDK for .NET: Delete search indexes
- Tablestore SDK for PHP: Delete search indexes

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.

Examples

```
private static void deleteSearchIndex(SyncClient client) {
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest();
    request.setTableName(TABLE_NAME); // Set the name of the table.
    request.setIndexName(INDEX_NAME); // Set the name of the search index.
    client.deleteSearchIndex(request); // Call client to delete the search Index.
}
```

6.6.7. Sorting and pagination

You can specify IndexSort when you create a search index and specify a sorting method when you query data. You can use limit and offset or tokens for pagination.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement sorting and pagination:

- Tablestore SDK for Java: Sorting and pagination
- Tablestore SDK for Go: Sorting and pagination
- Tablestore SDK for Python: Sorting and pagination
- Tablestore SDK for Node.js: Sorting and pagination
- Tablestore SDK for .NET: Sorting and pagination
- Tablestore SDK for PHP: Sorting and pagination

Index presorting

By default, data in a search index is sorted based on the value of the IndexSort parameter. When you use a search index to query data, the value of the IndexSort parameter determines the default order in which the matched data is returned.

When you create a search index, you can specify a value for the IndexSort parameter. If you do not specify a value for the IndexSort parameter, data in the search index is sorted by primary key. If you do not specify a value for the IndexSort parameter and use the search index to query data, the matched data is returned in the order of the primary key by default.

Notice Search indexes that contain fields of the Nested type do not support index presorting.

Specify a sorting method

Sorting can be enabled only for fields for which enableSortAndAgg is set to true.

You can specify a sorting method for each query. Search index-based queries support the following sorting methods. You can also specify multiple sorting methods based on different priorities.

- ScoreSort

  You can use ScoreSort to sort the query results based on the BM25-based keyword relevance score. ScoreSort is suitable for scenarios such as full-text search.

  Notice You must specify a value for ScoreSort to sort the matched data by keyword relevance score. Otherwise, the matched data is sorted based on the value that is specified for the IndexSort parameter.

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(new ScoreSort())));
  ```

- PrimaryKeySort

  You can use PrimaryKeySort to sort the query results based on the value of the primary key.

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(new PrimaryKeySort()))); // Sort the query res
  ults in the ascending order.
  //searchQuery.setSort(new Sort(Arrays.asList(new PrimaryKeySort(SortOrder.DESC)))); // So
  rt the query results in the descending order.
  ```

- FieldSort

  You can use FieldSort to sort the query results based on the values of a specified column.

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(new FieldSort("col", SortOrder.ASC))));
  ```

  You can also sort values in two columns in specified orders to determine the order in which the matched data is returned.

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(
      new FieldSort("col1", SortOrder.ASC), new FieldSort("col2", SortOrder.ASC))));
  ```

- GeoDistanceSort

You can use GeoDistanceSort to sort the query results by geographical location.

```
SearchQuery searchQuery = new SearchQuery();
// Sort the results based on the distance from the value in the GEOPOINT geo column to th
e coordinate pairs (0, 0).
Sort.Sorter sorter = new GeoDistanceSort("geo", Arrays.asList("0, 0"));
searchQuery.setSort(new Sort(Arrays.asList(sorter)));
```

Specify a pagination method

You can use the limit and offset parameters or use tokens to paginate the returned rows.

- Use the limit and offset parameters

  When the total number of returned rows to obtain is smaller than 50,000, you can configure the limit and offset parameters to paginate the rows. The sum of the limit and offset parameter values cannot exceed 50,000. The maximum value of the limit parameter is 100.

  > ? **Note** For more information about how to set limit to a value greater than 100, see How do I increase the value of the limit parameter to 1000 when I call the Search operation of the search index feature to query data?.

  If you use the limit and offset parameters to paginate the rows but do not specify values, the default values are used. The default value of the limit parameter is 10. The default value of the offset parameter is 0.

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setQuery(new MatchAllQuery());
  searchQuery.setLimit(100);
  searchQuery.setOffset(100);
  ```

- Use a token

  We recommend that you use a token for deep pagination because this method has no limits on the pagination depth.

  If Tablestore cannot read all data that meets the query conditions, Tablestore returns nextToken. You can use nextToken to continue to read the subsequent data.

  By default, you can only page backward when you use a token. However, you can cache and use the previous token to page forward because a token is valid during the query.

  > **Notice** If you need to persist nextToken or transfer nextToken to the frontend page, you can use Base64 to encode nextToken into a string. Tokens are not strings. If you use `new String(nextToken)` to encode a token into a string, information about the token is lost.

  When you use a token for pagination, the sorting method is the same as the method that is used in the previous request. Therefore, you cannot specify the sorting method if you use a token. You cannot set the offset parameter when a token is used. Data is returned page by page in sequence, which results in a slow query.

Notice Search indexes that contain fields of the Nested type do not support IndexSort. If you use a search index that contains fields of the Nested type to query data and require pagination, you must specify the sorting method in the query conditions to return data in the specified order. Otherwise, Tablestore does not return nextToken when only part of data that meets the query conditions is returned.

```
private static void readMoreRowsWithToken(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    searchQuery.setQuery(new MatchAllQuery());
    searchQuery.setGetTotalCount(true);// Specify that the total number of matched rows i
s returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    SearchResponse resp = client.search(searchRequest);
    if (!resp.isAllSuccess()) {
        throw new RuntimeException("not all success");
    }
    List<Row> rows = resp.getRows();
    while (resp.getNextToken()!=null) { // If nextToken is null in the response, all data
is read.
        // Query the nextToken value.
        byte[] nextToken = resp.getNextToken();
        {
            // If you need to persist nextToken or transfer nextToken to the frontend pag
e, you can use Base64 to encode nextToken into a string.
            // Tokens are not strings. If you use new String(nextToken) to encode a token
into a string, information about the token is lost.
            String tokenAsString = Base64.toBase64String(nextToken);
            // Decode the string into bytes.
            byte[] tokenAsByte = Base64.fromBase64String(tokenAsString);
        }
        // Set the token in this request to the nextToken value in the previous response.

        searchRequest.getSearchQuery().setToken(nextToken);
        resp = client.search(searchRequest);
        if (!resp.isAllSuccess()) {
            throw new RuntimeException("not all success");
        }
        rows.addAll(resp.getRows());
    }
    System.out.println("RowSize: " + rows.size());
    System.out.println("TotalCount: " + resp.getTotalCount());// Specify that the total n
umber of matched rows is displayed. The number of returned rows is not displayed.
}
```

6.6.8. Tokenization

After a tokenization method is specified for TEXT fields, Tablestore tokenizes field values into multiple tokens based on the tokenization method that you configure. You cannot specify tokenization methods for non-TEXT fields.

You can use match query (MatchQuery) and match phrase query (MatchPhraseQuery) to query TEXT data. You can also use term query (TermQuery), terms query (TermsQuery), prefix query (PrefixQuery), and wildcard query (WildcardQuery) based on your business scenario.

Tokenization methods

The following tokenization methods are supported:

- Single-word tokenization (SingleWord)

  This tokenization method applies to all natural languages such as Chinese, English, and Japanese. By default, the tokenization method for TEXT fields is single-word tokenization.

  After single-word tokenization is specified, Tablestore performs tokenization based on the following rules:

  - Chinese texts are tokenized based on each Chinese character. For example, "杭州" is tokenized into "杭" and "州". You can use match query (MatchQuery) or match phrase query (MatchPhraseQuery) and set the keyword to "杭" to query the data that contains "杭州".
  - Letters or digits are tokenized based on spaces or punctuation marks. Uppercase letters are converted to lowercase letters. For example, "Hang Zhou" is tokenized into "hang" and "zhou". You can use match query (MatchQuery) or match phrase query (MatchPhraseQuery) and set the keyword to "hang", "HANG", or "Hang" to query the rows that contain "Hang Zhou".
  - By default, alphanumeric characters such as model numbers are also tokenized by spaces or punctuation marks. However, these characters cannot be tokenized into smaller words. For example, "IPhone6" can only be tokenized into "IPhone6". When you use match query (MatchQuery) or match phrase query (MatchPhraseQuery), you must specify "iphone6". No results are returned if you use "iphone".

  The following table describes the parameters for single-word tokenization.

Parameter	Description
caseSensitive	Specifies whether to enable case sensitivity. Default value: false. If you set the parameter to false, all letters are converted to lowercase letters. If you do not need Tablestore to convert letters to lowercase letters, you can set the parameter to true.
delimitWord	Specifies whether to tokenize alphanumeric characters. Default value: false. You can set the delimitWord parameter to true to separate letters from digits. This way, "iphone6" is tokenized into "iphone" and "6".

- Delimiter tokenization (Split)

  Tablestore provides general dictionary-based tokenization. However, some industries require custom dictionaries for tokenization. To meet this requirement, Tablestore provides delimiter tokenization. You can perform tokenization by using custom methods, use delimiter tokenization, and then write data to Tablestore.

  Delimiter tokenization applies to all natural languages such as Chinese, English, and Japanese.

After the tokenization method is set to delimiter tokenization, the system tokenizes field values based on the specified delimiter. For example, a field value is "badminton,ping pong,rap". The delimiter is set to a comma (,). The value is tokenized into "badminton", "ping pong", and "rap". The field is indexed. When you use match query (MatchQuery) or match phrase query (MatchPhraseQuery) to query "badminton", "ping pong", "rap", or "badminton,ping pong", the row can be obtained.

The following table describes the parameters for delimiter tokenization.

Parameter	Description
delimiter	The delimiter. By default, the value is a whitespace character. You can customize the delimiter. ○ When you create a search index, the delimiter specified for field tokenization must be the same as the delimiter that is included in the value of the column in the data table. Otherwise, data may not be obtained. ○ If the custom delimiter is a number sign (#), replace the delimiter with an escape character.

- Minimum semantic unit-based tokenization (MinWord)

  This tokenization method applies to the Chinese language in full-text search scenarios.

  After the tokenization method is set to minimum semantic unit-based tokenization, Tablestore tokenizes the TEXT field values into the minimum number of semantic units when Tablestore performs a query.

- Maximum semantic unit-based tokenization (MaxWord)

  This tokenization method applies to the Chinese language in full-text search scenarios.

  After the tokenization method is set to maximum semantic unit-based tokenization, Tablestore tokenizes the TEXT field values into the maximum number of semantic units when Tablestore performs a query. However, different semantic units may overlap. The total length of the tokenized words is longer than the length of the original text. The index size is increased.

  This tokenization method can generate more tokens and increase the probability that the rows are matched. However, the index size is greatly increased. You can use match phrase query (MatchPhraseQuery) when the tokenization method is set to maximum semantic unit-based tokenization. Match query (MatchQuery) is more suitable for this tokenization method. If you use match phrase query together with this tokenization method, data may not be obtained due to overlapping tokens because the keyword is also tokenized based on maximum semantic unit-based tokenization.

- Fuzzy tokenization

  This tokenization method applies to all natural languages such as Chinese, English, and Japanese in scenarios that involve short text content, such as titles, movie names, book titles, file names, and directory names.

  The combination of fuzzy tokenization and match phrase query can be used to return query results at a low latency. The combination of fuzzy tokenization and match phrase query outperforms wildcard query (WildcardQuery). However, the index size is greatly increased.

  After the tokenization method is set to fuzzy tokenization, Tablestore performs tokenization by using n-gram. The results are between minChars and maxChars. For example, this tokenization method is used to populate the drop-down list.

Fuzzy tokenization converts the field values into lowercase letters. Therefore, fuzzy tokenization is case-insensitive and is similar to the LIKE operator in SQL.

To perform a fuzzy query, you must perform a match phrase query (MatchPhraseQuery) on the columns for which fuzzy tokenization is used. If you have more query requirements on the column, use the virtual column feature. For more information about the virtual column feature, see Virtual columns.

- Limits
  - You can use fuzzy tokenization to tokenize the TEXT field values that are equal to or smaller than 1,024 characters in length. If the TEXT field value exceeds 1,024 characters in length, Tablestore truncates the excess characters and discards them, and only tokenizes the first 1,024 characters.
  - To prevent excessive increase of index data, the difference between the values of maxChars and minChars must not exceed 6.
- Parameters

Parameter	Description
minChars	The minimum number of characters for a token. Default value: 1.
maxChars	The maximum number of characters for a token. Default value: 7.

Comparison

The following table compares the tokenization methods.

Dimension	Single-word tokenization	Delimiter tokenization	Minimum semantic unit-based tokenization	Maximum semantic unit-based tokenization	Fuzzy tokenization
Index increase	Small	Small	Small	Medium	Large
Relevance	Weak	Weak	Medium	Relatively strong	Relatively strong
Applicable language	All	All	Chinese	Chinese	All
Length limit	None	None	None	None	1,024 characters
Recall rate	High	Low	Low	Medium	Medium

6.6.9. Match all query

You can use match all query to match all rows in a table to query the total number of rows in the table and return multiple random rows.

API operation

You can call the Search or ParallelScan operation and set the query type to MatchAllQuery to perform a match all query.

Usage

You can use the following Tablestore SDKs to perform a match all query:

- Tablestore SDK for Java: Match all query
- Tablestore SDK for Go: Match all query
- Tablestore SDK for Python: Match all query
- Tablestore SDK for Node.js: Match all query
- Tablestore SDK for .NET: Match all query
- Tablestore SDK for PHP: Match all query

Parameters

Parameter	Description
query	The query type. Set the query type to MatchAllQuery.
tableName	The name of the data table.
indexName	The name of the search index.
limit	The maximum number of rows that you want the current query to return. If you want the current query to return multiple random rows of data, set limit to a positive integer. To query only the number of matched rows without returning specific data, you can set limit to 0. This way, Tablestore returns the number of matched rows without data from the table.
columnsToGet	Specifies whether to return all columns. By default, returnAll is false, which specifies that not all columns are returned. In this case, you can use columns to specify the columns that you want to return. If you do not specify the columns that you want to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, getTotalCount is false, which specifies that the total number of rows that match the query conditions is not returned. If this parameter is set to true, the query performance is compromised.

Examples

```
/**
 * Use match all query to query the total number of rows in a table.
 * @param client
 */
private static void matchAllQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    /**
    * Set the query type to MatchAllQuery.
    */
    searchQuery.setQuery(new MatchAllQuery());
    /**
     * In the MatchAllQuery-based query result, the value of TotalCount is the total number
of rows in the table.
     If you want the current query to return multiple random rows of data, set limit to a p
ositive integer.
     * To query only the number of matched rows without returning specific data, you can se
t limit to 0. This way, Tablestore returns the number of matched rows without data from the
table.
     */
    searchQuery.setLimit(0);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME, searchQuery);
    /**
    * Specify that the total number of rows that match the query conditions is returned.
    */
    searchQuery.setGetTotalCount(true);
    SearchResponse resp = client.search(searchRequest);
    /**
     * Check whether the returned total number of rows that match the query conditions is c
orrect. If isAllSuccess is false, Tablestore may fail to query data on all servers and retu
rn a value that is smaller than the total number of rows that match the query conditions.
     */
    if (!resp.isAllSuccess()) {
        System.out.println("NotAllSuccess!");
    }
    System.out.println("IsAllSuccess: " + resp.isAllSuccess());
    System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that the total nu
mber of rows that match the query conditions is displayed.
    System.out.println(resp.getRequestId());
}
```

6.6.10. Match query

You can use match query (MatchQuery) to query data in a table based on approximate matches. Tablestore tokenizes the values in TEXT columns and the keywords you use to perform match queries based on the analyzer that you specify. Therefore, Tablestore can perform match queries based on the tokens. We recommend that you use match phase query (MatchPhraseQuery) for columns for which fuzzy tokenization is used to ensure high performance in fuzzy queries.

Scenarios

You can use match query to query data in TEXT columns in full-text search scenarios. For example, the value in the title column of a row is "Hangzhou West Lake Scenic Area" and single-word tokenization is used. If you set the keyword to "Lake Scenic" for the match query, Tablestore returns this row in the query result.

API operation

You can call the Search or ParallelScan operation and set the query type to MatchQuery to perform a match query.

Usage

You can use the following Tablestore SDKs to perform a match query:

- Tablestore SDK for Java: Match query
- Tablestore SDK for Go: Match query
- Tablestore SDK for Python: Match query
- Tablestore SDK for Node.js: Match query
- Tablestore SDK for .NET: Match query
- Tablestore SDK for PHP: Match query

Parameters

Parameter	Description
fieldName	The name of the column that you want to query. Match query applies to TEXT columns.
text	The keyword that is used to match the column values when you perform a match query. If the column to query is a TEXT column, the keyword is tokenized into multiple tokens based on the analyzer that you specify when you create the search index. By default, single-word tokenization is performed if you do not specify the analyzer when you create the search index. For example, if you set the tokenization method to single-word tokenization and use "this is" as a search keyword, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
query	The query type, which is set to matchQuery.
offset	The position from which the current query starts.

Parameter	Description
limit	The maximum number of rows that you want the current query to return. To query only the number of matched rows without returning specific data, you can set limit to 0. This way, Tablestore returns the number of matched rows instead of specific data from the table.
minimumShouldMatch	The minimum number of matched tokens contained in a column value. A row is returned only when the value of the fieldName column in the row contains at least the minimum number of matched tokens. **Note** minimumShouldMatch must be used together with the OR logical operator.
operator	The logical operator. By default, OR is used as the logical operator, which specifies that a row matches the query conditions when one of the tokens is matched. If you set the operator to AND, the row meets the query conditions only when the column value contains all tokens.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, the value of this parameter is false, which specifies that the total number of rows that match the query conditions is not returned. If this parameter is set to true, the query performance is compromised.
tableName	The name of the data table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, the value of returnAll is false, which specifies that not all columns are returned. In this case, you can use columns to specify the columns that you want to return. If you do not specify the columns that you want to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows in which the value in Col_Keyword matches "hangzhou". Tablestore returns part of the matched rows and the total number of matched rows in this query.
 * @param client
 */
private static void matchQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    MatchQuery matchQuery = new MatchQuery(); // Set the query type to MatchQuery.
    matchQuery.setFieldName("Col_Keyword"); // Specify the name of the column that you want to query.
    matchQuery.setText("hangzhou"); // Specify the keyword that you want to match.
    searchQuery.setQuery(matchQuery);
    searchQuery.setOffset(0); // Set offset to 0.
    searchQuery.setLimit(20); // Set limit to 20 to return up to 20 rows.
    //searchQuery.setGetTotalCount(true); // Specify that the total number of matched rows is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can configure the columnsToGet parameter to specify the columns to return or specify that all columns are returned. If you do not configure this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns to return the columns that you want to return.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that the total number of matched rows instead of the number of returned rows is displayed.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.11. Match phrase query

Match phrase query is similar to match query, except match phrase query evaluates the positions of tokens. A row meets the query condition only when the order and positions of the tokens in the row match the order and positions of the tokens that are contained in the keyword. If the tokenization method for the column that you want to query is fuzzy tokenization, match phrase query is performed at a lower latency than wildcard query.

For example, the value in the column is "Hangzhou West Lake Scenic Area" and the keyword you specify in Query is "Hangzhou Scenic Area". Tablestore returns the row when you use match query. However, when you use match phrase query, Tablestore does not return the row. The distance between "Hangzhou" and "Scenic Area" in Query is 0, but the distance in the column of this row is 2 because the two words "West" and "Lake" exist between "Hangzhou" and "Scenic Area".

API operation

You can call the Search or ParallelScan operation and set the query type to MatchPhraseQuery to perform a match phrase query.

Usage

You can use the following Tablestore SDKs to perform a match phrase query:

- Tablestore SDK for Java: Match phrase query
- Tablestore SDK for Go: Match phrase query
- Tablestore SDK for Python: Match phrase query
- Tablestore SDK for Node.js: Match phrase query
- Tablestore SDK for .NET: Match phrase query
- Tablestore SDK for PHP: Match phrase query

Parameters

Parameter	Description
fieldName	The name of the column that you want to query. Match phrase query applies to TEXT columns.
text	The keyword that is used to match the column values when you perform a match phrase query. If the column to query is a TEXT column, the keyword is tokenized into multiple tokens based on the analyzer that you specify when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you query the phrase "this is", "..., this is tablestore" and "this is a table" are returned. "this table is ..." and "is this a table" are not returned.
query	The query type, which is set to matchPhraseQuery.
offset	The position from which the current query starts.
limit	The maximum number of rows that you want the current query to return. To query only the number of matched rows without returning specific data, you can set limit to 0. This way, Tablestore returns the number of matched rows instead of specific data from the table.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, the value of this parameter is false, which specifies that the total number of rows that match the query conditions is not returned. If you set this parameter to true, the query performance is compromised.

Parameter	Description
tableName	The name of the data table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, the value of returnAll is false, which specifies that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns that you want to return. If you do not specify the columns that you want to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows in which the value of the Col_Text column matches the whole ph
ase "hangzhou shanghai" in order. Tablestore returns part of the matched rows and the numbe
r of rows that match the phrase in this query.
 * @param client
 */
private static void matchPhraseQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    MatchPhraseQuery matchPhraseQuery = new MatchPhraseQuery(); // Set the query type to Ma
tchPhraseQuery.
    matchPhraseQuery.setFieldName("Col_Text"); // Specify the name of the column to query.
    matchPhraseQuery.setText("hangzhou shanghai"); // Specify the keyword that you want to
match.
    searchQuery.setQuery(matchPhraseQuery);
    searchQuery.setOffset(0); // Set offset to 0.
    searchQuery.setLimit(20); // Set limit to 20 to return up to 20 rows.
    //searchQuery.setGetTotalCount(true); // Specify that the total number of matched rows
is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    // You can configure the columnsToGet parameter to specify the columns that you want to
return or specify that all columns are returned. If you do not configure this parameter, on
ly the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns to retu
rn specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that the total
number of matched rows instead of the number of returned rows is displayed.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.12. Term query

You can use term query to query data that exactly matches the specified value of a field. Term query is similar to queries based on string match conditions. If the type of a field is TEXT, Tablestore tokenizes the string and exactly matches tokens.

Operations

You can call the Search or ParallelScan operation and set the query type to TermQuery to implement term query.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement term query:

- Tablestore SDK for Java: Term query
- Tablestore SDK for Go: Term query
- Tablestore SDK for Python: Term query
- Tablestore SDK for Node.js: Term query
- Tablestore SDK for .NET: Term query
- Tablestore SDK for PHP: Term query

Parameters

Parameter	Description
query	The type of the query. Set the query type to TermQuery.
fieldName	The name of the field you want to match.
term	The keyword used to match the column values when you perform a term query. This word is not tokenized. Instead, the whole word is used to match the field values. If the type of a field is TEXT, Tablestore tokenizes the string and exactly matches tokens. For example, TEXT string "tablestore is cool" is tokenized into "tablestore", "is", and "cool". When you specify one of these tokens as a search string, you can retrieve query results that contain "tablestore is cool".
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
tableName	The name of the table.
indexName	The name of the search index.

Parameter	Description
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword exactly matches "hangzhou".
 * @param client
 */
private static void termQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermQuery termQuery = new TermQuery(); // Set the query type to TermQuery.
    termQuery.setFieldName("Col_Keyword"); // Set the name of the field that you want to match.
    termQuery.setTerm(ColumnValue.fromString("hangzhou")); // Set the value that you want to match.
    searchQuery.setQuery(termQuery);
    //searchQuery.setGetTotalCount(true); // Specify that the total number of matched rows is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can set the columnsToGet parameter to specify the columns to return or specify that all columns are returned. If you do not set this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total number of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.13. Terms query

This query is similar to a term query. A terms query supports multiple terms. A row of data is returned if one of the keywords matches field values. Terms queries can be used in the same manner as the IN operator in SQL statements.

Operations

You can call the Search or ParallelScan operation and set the query type to TermsQuery to perform a terms query.

Use Tablestore SDKs

You can use the following Tablestore SDKs to perform terms queries:

- Tablestore SDK for Java: Terms query
- Tablestore SDK for Go: Terms query
- Tablestore SDK for Python: Terms query
- Tablestore SDK for Node.js: Terms query
- Tablestore SDK for PHP: Terms query

Parameters

Parameter	Description
query	The type of the query. Set the value to TermsQuery.
fieldName	The name of the field that you want to match.
terms	The keywords that are used to match the field values when you perform a terms query. You can specify up to 1,024 keywords. A row of data is returned if one of the keywords matches field values.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false. This value indicates that the total number of rows that match the query conditions is not returned. If this parameter is set to true, query performance is affected.
tableName	The name of the data table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure ReturnAll and Columns for this parameter. By default, ReturnAll is set to false. This value indicates that not all columns of each matched row are returned. If ReturnAll is set to false, you can use Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns of each matched row are returned.

Examples

```
/**
 * Search the table for rows in which the value of Col_Keyword is "hangzhou" or "xi'an".
 * @param client
 */
private static void termQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermsQuery termsQuery = new TermsQuery(); // Set the query type to TermsQuery.
    termsQuery.setFieldName("Col_Keyword"); // Specify the name of the field that you want to match.
    termsQuery.addTerm(ColumnValue.fromString("hangzhou")); // Specify the keyword that you want to match.
    termsQuery.addTerm(ColumnValue.fromString("xi'an")); // Specify the keyword that you want to match.
    searchQuery.setQuery(termsQuery);
    //searchQuery.setGetTotalCount(true); // Specify that the total number of matched rows is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can configure the columnsToGet parameter to specify the columns to return or specify that all columns are returned. If you do not configure this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Specify the columns to return.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that the total number of matched rows but not returned rows is displayed.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.14. Prefix query

You can use prefix query to query data that matches a specified prefix. If the type of a field is TEXT, Tablestore tokenizes the string and matches tokens by using the specified prefix.

Operations

You can call the Search or ParallelScan operation and set the query type to PrefixQuery to implement prefix query.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement prefix query:

- Tablestore SDK for Java: Prefix query
- Tablestore SDK for Go: Prefix query
- Tablestore SDK for Python: Prefix query
- Tablestore SDK for Node.js: Prefix query
- Tablestore SDK for .NET: Prefix query

- Tablestore SDK for PHP: Prefix query

Parameters

Parameter	Description
query	The type of the query. Set the query type to PrefixQuery.
fieldName	The name of the field that you want to match.
prefix	The prefix. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that contains the specified prefix.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword contains the prefix that exactl
y matches "hangzhou".
 * @param client
 */
private static void prefixQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    PrefixQuery prefixQuery = new PrefixQuery(); // Set the query type to PrefixQuery.
    searchQuery.setGetTotalCount(true);
    prefixQuery.setFieldName("Col_Keyword");
    prefixQuery.setPrefix("hangzhou");
    searchQuery.setQuery(prefixQuery);
    //searchQuery.setGetTotalCount(true); // Specify that the total number of matched rows
is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    // You can set the columnsToGet parameter to specify the columns to return or specify t
hat all columns are returned. If you do not set this parameter, only the primary key column
s are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns to retu
rn specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total numbe
r of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());
```

6.6.15. Range query

You can use RangeQuery to query data that falls within a specified range. When a table contains a TEXT string, Tablestore tokenizes the string and matches tokens by using the specified prefix.

Operations

You can call the Search or ParallelScan operation and set the query type to RangeQuery to perform range queries.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement range query:

- Tablestore SDK for Java: Range query
- Tablestore SDK for Go: Range query
- Tablestore SDK for Python: Range query
- Tablestore SDK for Node.js: Range query
- Tablestore SDK for .NET: Range query
- Tablestore SDK for PHP: Range query

Parameters

Parameter	Description
fieldName	The name of the field you want to match.
from	The value from which the query starts. When you set range conditions, you can use greaterThan to indicate the greater than (>) sign and greaterThanOrEqual to indicate the greater-than-or-equal-to (≥) sign.
to	The value with which the query ends. When you set range conditions, you can use lessThan to indicate the less-than (<) sign and lessThanOrEqual to indicate the less-than-or-equal-to (≤) sign.
includeLower	Specifies whether to include the value of the from parameter in the response. Type: Boolean.
includeUpper	Specifies whether to include the value of the to parameter in the response. Type: Boolean.
query	The query type, which is set to RangeQuery.
sort	The sorting method. For more information, see Sorting and pagination.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when the total number of rows that match the query conditions is returned.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows where the value in Col_Long is greater than 3. Tablestore sort
s these rows by Col_Long in descending order.
 * @param client
 */
private static void rangeQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    RangeQuery rangeQuery = new RangeQuery(); // Set the query type to RangeQuery.
    rangeQuery.setFieldName("Col_Long");  // Set the name of the field that you want to mat
ch.
    rangeQuery.greaterThan(ColumnValue.fromLong(3));  // Specify the range of the values of
the field. The matched values are greater than 3.
    searchQuery.setGetTotalCount(true);
    searchQuery.setQuery(rangeQuery);
    // Sort the query results by Col_Long in descending order.
    FieldSort fieldSort = new FieldSort("Col_Long");
    fieldSort.setOrder(SortOrder.DESC);
    searchQuery.setSort(new Sort(Arrays.asList((Sort.Sorter)fieldSort)));
    //searchQuery.setGetTotalCount(true);//Set the total number of matched rows to return.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    // You can set the columnsToGet parameter to specify the columns to return or specify t
o return all columns. If you do not set this parameter, only the primary key columns are re
turned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set returnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set columns to retu
rn specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total numbe
r of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.16. Wildcard query

When you perform a wildcard query, you can use the asterisk (*) and question mark (?) wildcard characters in the query to search for data. The asterisk (*) matches a string of any length at, before, or after a search term. The question mark (?) matches a single character in a specific position. The string can start with an asterisk (*) or a question mark (?). For example, if you search for the "table*e" string, "tablestore" can be matched.

The `*word*` string is equivalent to the `WHERE field_a LIKE '%word%'` clause in SQL. If you want to search for the *word* string, you can perform a fuzzy query that provides higher performance than a wildcard query. For more information about how to perform a fuzzy query, see Fuzzy query. If you perform a fuzzy query, the query performance is not compromised when the data volume increases.

Limit

A string that contains wildcards can be up to 32 characters in length.

API operations

You can call the Search or ParallelScan operation and set the query type to WildcardQuery to perform a wildcard query.

Tablestore SDKs

You can use the following Tablestore SDKs to perform a wildcard query:

- Tablestore SDK for Java: Wildcard query
- Tablestore SDK for Go: Wildcard query
- Tablestore SDK for Python: Wildcard query
- Tablestore SDK for Node.js: Wildcard query
- Tablestore SDK for .NET: Wildcard query
- Tablestore SDK for PHP: Wildcard query

Parameters

Parameter	Description
fieldName	The name of the column.
value	The string that contains wildcards. The string cannot exceed 32 characters in length.
query	The type of the query. Set the query type to WildcardQuery.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. The default value of this parameter is false, which indicates that the total number of rows that match the query conditions is not returned. If this parameter is set to true, the query performance is compromised.
tableName	The name of the data table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each row that meets the query conditions. You can configure returnAll and columns for this parameter. The default value of returnAll is false, which indicates that not all columns are returned. In this case, you can use columns to specify the columns that you want to return. If you do not specify the columns that you want to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows in which the value of the Col_Keyword column matches "hang*u".

 * @param client
 */
private static void wildcardQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    WildcardQuery wildcardQuery = new WildcardQuery(); // Set the query type to WildcardQuery.
    wildcardQuery.setFieldName("Col_Keyword");
    wildcardQuery.setValue("hang*u"); // Specify a string that contains one or more wildcard characters in wildcardQuery.
    searchQuery.setQuery(wildcardQuery);
    //searchQuery.setGetTotalCount(true); // Specify that the total number of rows that meet the query conditions is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can use the columnsToGet parameter to specify the columns that you want to return or specify that all columns are returned. If you do not specify this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Specify the columns that you want to return.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that the total number of rows that meet the query conditions instead of the number of returned rows is displayed.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.17. Boolean query

This topic describes how to use Boolean query to query the rows based on a combination of subqueries. Tablestore returns the rows that match the subqueries. Each subquery can be of any type, including BoolQuery.

API operations

You can call the Search or ParallelScan operation and set the query type to BoolQuery to implement Boolean queries.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement Boolean queries:

- Tablestore SDK for Java: Boolean query
- Tablestore SDK for Go: Boolean query
- Tablestore SDK for Python: Boolean query
- Tablestore SDK for Node.js: Boolean query
- Tablestore SDK for .NET: Boolean query

- Tablestore SDK for PHP: Boolean query

Parameters

Parameter	Description
mustQueries	The list of subqueries that the query results must match. This parameter is equivalent to the AND operator.
mustNotQueries	The list of subqueries that the query results must not match. This parameter is equivalent to the NOT operator.
filterQueries	The list of subqueries. Only rows that match all subfilters are returned. filter is similar to query except that filter does not calculate the relevance score based on the number of subfilters that the row matches.
shouldQueries	The list of subqueries that the query results can or cannot match. This parameter is equivalent to the OR operator. Only rows that meet the minimum number of subquery conditions specified by shouldQueries are returned. A higher overall relevance score indicates that more subquery conditions specified by shouldQueries are met.
minimumShouldMatch	The minimum number of subquery conditions specified by shouldQueries that the rows must meet. If no other subquery conditions except the subquery conditions that are specified by shouldQueries are specified, the default value of the minimumShouldMatch parameter is 1. If other subquery conditions, such as subquery conditions specified by mustQueries, mustNotQueries, and filterQueries are specified, the default value of the minimumShouldMatch parameter is 0.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, getTotalCount is false, which indicates that the total number of rows that match the query conditions is not returned. If this parameter is set to true, query performance is compromised.
tableName	The name of the data table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is false, which indicates that not all columns are returned. In this case, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

- Example 1:

  Perform a Boolean query to query the rows that match the AND-based conditions.

```
/**
 * Perform a Boolean query to query the rows that match the AND-based conditions.
 * @param client
 */
public static void andQuery(SyncClient client){
    /**
     * Condition 1: Perform a range query to query the rows where the Col_Long column val
ue is greater than 3.
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));
    /**
     * Condition 2: Perform a match query to query the rows where the Col_Keyword column
value matches hangzhou.
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * Construct a Boolean query where the query results meet both Condition 1 and Co
ndition 2.
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustQueries(Arrays.asList(rangeQuery, matchQuery));
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);// Set GetTotalCount to true to return the t
otal number of matched rows.
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        // You can configure the columnsToGet parameter to specify the columns to return
or specify that all columns are returned. If you do not configure this parameter, only th
e primary key columns are returned.
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns
.
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns t
o return specified columns.
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse resp = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that all
rows that match the specified conditions are displayed. The number of rows to return is n
ot displayed.
        System.out.println("Row: " + resp.getRows());
    }
}
```

- Example 2

  Perform a Boolean query to query the rows that match the OR-based conditions.

```
/**
 * Perform a Boolean query to query the rows that match the OR-based conditions.
 * @param client
 */
public static void orQuery(SyncClient client) {
    /**
     * Condition 1: Perform a range query to query the rows where the Col_Long column value is greater than 3.
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));
    /**
     * Condition 2: Perform a match query to query the rows where the Col_Keyword column value matches hangzhou.
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
    /**
     * Construct a Boolean query where the query results meet at least one of Condition 1 and Condition 2.
     */
    BoolQuery boolQuery = new BoolQuery();
    boolQuery.setShouldQueries(Arrays.asList(rangeQuery, matchQuery));
    boolQuery.setMinimumShouldMatch(1); // Specify that the results meet at least one of the conditions.
    searchQuery.setQuery(boolQuery);
    //searchQuery.setGetTotalCount(true);// Specify that the total number of matched rows is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can configure the columnsToGet parameter to specify the columns to return or specify that all columns are returned. If you do not configure this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that all rows that match the specified conditions are displayed. The number of rows to return is not displayed.
    System.out.println("Row: " + resp.getRows());
    }
}
```

- Example 3

  Perform a Boolean query to query the rows that match the NOT-based conditions.

```
/**
 * Perform a Boolean query to query the rows that match the NOT-based conditions.
 * @param client
 */
public static void notQuery(SyncClient client) {
    /**
     * Condition 1: Perform a match query to query the rows where the Col_Keyword column
value matches hangzhou.
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * Construct a Boolean query where the query results do not meet Condition 1.
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustNotQueries(Arrays.asList(matchQuery));
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);// Specify that the total number of matched
rows is returned.
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        // You can set the columnsToGet parameter to specify the columns to return or spe
cify that all columns are returned. If you do not set this parameter, only the primary ke
y columns are returned.
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns
.
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns t
o return specified columns.
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse resp = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that all
rows that match the specified conditions are displayed. The number of rows to return is n
ot displayed.
        System.out.println("Row: " + resp.getRows());
    }
}
```

- Example 4

  The following sample code provides an example on how to perform a Boolean query that includes multiple subqueries of the BoolQuery type. In (col2 < 4 or col3 < 5) or (col2 = 4 and (col3 = 5 or col3 = 6)), each subquery of the BoolQuery type is connected by AND or OR.

```
/**
 * (col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6))
 * In the preceding example, each subquery of the BoolQuery type is connected by AND or O
R.
 * @param client
 * @param client
*/
private static void boolQuery2(SyncClient client){
        // Condition 1: col2 < 4
        RangeQuery rangeQuery1 = new RangeQuery();
        rangeQuery1.setFieldName("col2");
        rangeQuery1.lessThan(ColumnValue.fromLong(4));
        // Condition 2: col3 < 5
        RangeQuery rangeQuery2 = new RangeQuery();
        rangeQuery2.setFieldName("col3");
        rangeQuery2.lessThan(ColumnValue.fromLong(5));
        // Condition 3: col2 = 4
        TermQuery termQuery = new TermQuery();
        termQuery.setFieldName("col2");
        termQuery.setTerm(ColumnValue.fromLong(4));
        // Condition 4: col3 = 5 or col3 = 6
        TermsQuery termsQuery = new TermsQuery();
        termsQuery.setFieldName("col3");
        termsQuery.addTerm(ColumnValue.fromLong(5));
        termsQuery.addTerm(ColumnValue.fromLong(6));
        SearchQuery searchQuery = new SearchQuery();
        List<Query> queryList1 = new ArrayList<>();
        queryList1.add(rangeQuery1);
        queryList1.add(rangeQuery2);
        // Combination 1: col2 < 4 or col3 < 5
        BoolQuery boolQuery1 = new BoolQuery();
        boolQuery1.setShouldQueries(queryList1);
        // Combination 2: col2 = 4 and (col3 = 5 or col3 = 6)
        List<Query> queryList2 = new ArrayList<>();
        queryList2.add(termQuery);
        queryList2.add(termsQuery);
        BoolQuery boolQuery2 = new BoolQuery();
        boolQuery2.setMustQueries(queryList2);
        // Final combination: (col2 < 4 or col3 < 5) or (col2 = 4 and (col3 = 5 or col3 =
6))
        List<Query> queryList3 = new ArrayList<>();
        queryList3.add(boolQuery1);
        queryList3.add(boolQuery2);
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setShouldQueries(queryList3);
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);// Specify that the total number of matched
rows is returned.
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        // You can configure the columnsToGet parameter to specify the columns to return
or specify that all columns are returned. If you do not configure this parameter, only th
e primary key columns are returned.
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns
.
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns t
o return specified columns.
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse response = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that all rows that match the specified conditions are displayed. The number of rows to return is not displayed.
        System.out.println(response.getRows());
    }
```

6.6.18. Nested query

This topic describes how to use nested query to query the data in the child rows of nested fields. Nested fields cannot be directly queried. To query a nested field, you must specify the path of the nested field and a subquery in a NestedQuery object. The subquery can be a query of any type.

Note

- Only nested columns can be queried in nested queries.
- However, you can implement nested queries and queries of other types in a single request. For more information about the nested type, see Nested.

Operations

You can call the Search or ParallelScan operation and set the query type to NestedQuery to implement nested queries.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement nested queries:

- Tablestore SDK for Java: Nested query
- Tablestore SDK for Go: Nested query
- Tablestore SDK for Python: Nested query
- Tablestore SDK for Node.js: Nested query
- Tablestore SDK for .NET: Nested query
- Tablestore SDK for PHP: Nested query

Parameters

Parameter	Description
path	The path of the nested field. The path is similar to the tree structure. For example, news.title indicates the title subcolumn in the nested field named news.
query	The query implemented on the subcolumn in the nested field. The query can be of any query type.
scoreMode	Specifies which value is used to calculate the score when the field contains multiple values.

Parameter	Description
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

The following examples show how to use nested queries.

- Example 1

  The following code provides an example on how to query data that matches the "tablestore" condition in the col_nested.nested_1 column. In this example, the nested field named col_nested includes two subcolumns: nested_1 and nested_2.

```
private static void nestedQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    NestedQuery nestedQuery = new NestedQuery(); // Set the query type to NestedQuery.
    nestedQuery.setPath("col_nested"); // Set the path of the nested column.
    TermQuery termQuery = new TermQuery(); // Create the child query of the NestedQuery o
bject.
    termQuery.setFieldName("col_nested.nested_1"); // Set the column name. Note the path
that includes nested columns.
    termQuery.setTerm(ColumnValue.fromString("tablestore")); // Specify the value that yo
u want to query.
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    //searchQuery.setGetTotalCount(true);// Specify the setGetTotalCount parameter to tru
e to return the total number of rows that match the query condition.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    // You can set the columnsToGet parameter to specify whether to return all columns or
only specified columns. By default, if this parameter is not set, only the primary column
s are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set the setReturnAll parameter to return all co
lumns.
```

```
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set the setColumns parameter to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: "+ resp.getTotalCount()); // Display the total number of matched columns but not returned columns.
System.out.println("Row:  " + resp.getRows());
}
    TermQuery termQuery = new TermQuery(); // Create the child query of the NestedQuery object.
    termQuery.setFieldName("col_nested.nested_1"); // Set the column name. Note the path that includes nested columns.
    termQuery.setTerm(ColumnValue.fromString("tablestore")); // Specify the value that you want to query.
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    //searchQuery.setGetTotalCount(true);// Specify the setGetTotalCount parameter to true to return the total number of rows that match the query condition.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can set the columnsToGet parameter to specify whether to return all columns or only specified columns. By default, if this parameter is not set, only the primary columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set the setReturnAll parameter to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set the setColumns parameter to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total number of matched columns but not returned columns.
    System.out.println("Row: " + resp.getRows());
}
```

- Example 2

  The following code provides an example on how to query data that matches the "tablestore" condition in the col_nested.nested_2.nested_2_2 column. In this example, the nested field named col_nested includes two subcolumns: nested_1 and nested_2. The nested_2 subcolumn includes two columns: nested_2_1 and nested_2_2.

```
private static void nestedQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    NestedQuery nestedQuery = new NestedQuery(); // Set the query type to NestedQuery.
    nestedQuery.setPath("col_nested.nested_2"); // Set the path of the nested column, whi
ch is the parent path of the field to query.
    TermQuery termQuery = new TermQuery(); // Create the child query of the NestedQuery o
bject.
    termQuery.setFieldName("col_nested.nested_2.nested_2_2"); // Set the path of the colu
mn, which is the full path of the field to query.
    termQuery.setTerm(ColumnValue.fromString("tablestore")); // Specify the value that yo
u want to query.
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    //searchQuery.setGetTotalCount(true);// Set the setGetTotalCount parameter to true to
return the total number of rows that match the query condition.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    // You can set the columnsToGet parameter to specify whether to return all columns or
only specified columns. By default, if this parameter is not set, only the primary column
s are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set the setReturnAll parameter to return all co
lumns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set the setColumn
s parameter to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total num
ber of matched columns but not returned columns.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.19. Geo-distance query

You can use geo-distance query to specify a circular geographical area that is defined by a central point and radius as a query condition. Tablestore returns the rows where the value of a field falls within the circular geographical area.

Parameters

Parameter	Description
fieldName	The name of the column. Set the query type to Geopoint.
centerPoint	The coordinate pair of the central point. The coordinate pair consists of latitude and longitude values. This parameter value must be in the format of `latitude,longitude`. Valid values of the latitude: [-90,+90]. Valid values of the longitude: [-180,+180]. Example: `35.8,-45.91`.

Parameter	Description
distanceInMeter	The radius of the circle centered on the specified location. Type: DOUBLE. Unit: meter.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
query	The query statement for the search index. Set the query type to GeoDistanceQuery.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

Search the table for rows where the value of Col_GeoPoint falls within a specified distance from a specified central point.

```
public static void geoDistanceQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoDistanceQuery geoDistanceQuery = new GeoDistanceQuery();  // Set the query type to GeoDistanceQuery.
    geoDistanceQuery.setFieldName("Col_GeoPoint");
    geoDistanceQuery.setCenterPoint("5,5"); // Specify the coordinate pair for a central point.
    geoDistanceQuery.setDistanceInMeter(10000); // You can set the distance from the central point to a value greater than or equal to 10,000. Unit: meter.
    searchQuery.setQuery(geoDistanceQuery);
    //searchQuery.setGetTotalCount(true); // Specify that the total number of matched rows is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can set the columnsToGet parameter to specify the columns to return or specify that all columns are returned. If you do not set this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set Columns to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total number of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.20. Geo-bounding box query

You can use geo-bounding box query to query data that falls within a rectangular geographic area. You can specify the rectangular geographic area as a query condition. Tablestore returns the rows where the value of a field falls within the rectangular geographic area.

Parameters

Parameter	Description
fieldName	The name of the column. Set the query type to Geopoint.
topLeft	The coordinate pair of the upper-left corner of the rectangular geographic area.
bottomRight	The coordinate pair of the lower-right corner of the rectangular geographic area. The coordinate pairs of the upper-left corner and lower-right corner define a unique rectangular geographic area. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90, 90]. Valid values of the longitude: [-180, 180]. Example: “35.8,-45.91".

Parameter	Description
query	The query statement for the search index. Set the query type to GeoBoundingBoxQuery.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when the total number of rows that match the query conditions is returned.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns you want to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

* The data type of Col_GeoPoint is GeoPoint. You can obtain the rows where the value of Col_GeoPoint falls within the rectangular geographic area where the upper-left corner is at "10,0" and the lower-right corner is at "0,10".

```
/**
 * @param client
 */public static void geoBoundingBoxQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoBoundingBoxQuery geoBoundingBoxQuery = new GeoBoundingBoxQuery(); // Set the query t
ype to GeoBoundingBoxQuery.
    geoBoundingBoxQuery.setFieldName("Col_GeoPoint"); // Set the name of the field that you
want to match.
    geoBoundingBoxQuery.setTopLeft("10,0"); // Specify coordinates for the upper-left corne
r of the rectangular geographic area.
    geoBoundingBoxQuery.setBottomRight("0,10"); // Specify coordinates for the lower-right
corner of the rectangular geographic area.
    searchQuery.setQuery(geoBoundingBoxQuery);
    //searchQuery.setGetTotalCount(true);//Set the total number of matched rows to return.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    // You can set the columnsToGet parameter to specify the columns to return or specify t
o return all columns. If you do not set this parameter, only the primary key columns are re
turned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set returnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set columns to retu
rn specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total numbe
r of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.21. Geo-polygon query

You can use geo-polygon query to query data that falls within a polygon geographic area. You can specify the polygon geographic area as a query condition. Tablestore returns the rows where the value of a field falls within the polygon geographic area.

Parameters

Parameter	Description
fieldName	The name of the column. Set the query type to Geopoint.
points	The coordinate pairs of the points that define a polygon area. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of longitude: [-180,180]. Example: “35.8,-45.91".
query	The query statement for the search index. Set the query type to GeoPolygonQuery.

Parameter	Description
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when the total number of rows that match the query conditions is returned.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

Query the table for rows where the value of Col_GeoPoint falls within a specified polygon geographic area.

```
public static void geoPolygonQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoPolygonQuery geoPolygonQuery = new GeoPolygonQuery();  // Set the query type to GeoPolygonQuery.
    geoPolygonQuery.setFieldName("Col_GeoPoint");
    geoPolygonQuery.setPoints(Arrays.asList("0,0","5,5","5,0")); // Specify coordinate pairs for vertices of a polygon geographic area.
    searchQuery.setQuery(geoPolygonQuery);
    //searchQuery.setGetTotalCount(true);//Set the total number of matched rows to return.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can set the columnsToGet parameter to specify the columns to return or specify to return all columns. If you do not set this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set returnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set columns to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total number of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.22. Exists query

Exists query is also called NULL query or NULL-value query. This query is used in sparse data to determine whether a column of a row exists. For example, you can query the rows in which the value of the address column is not empty.

Note

- If you want to check whether a column contains empty values, you must use ExistsQuery together with mustNotQueries of BoolQuery.
- If one of the following conditions is met, the system considers that a column does not to exist. In this example, the city column is used.
  - The type of the city column in the search index is a basic type such as keyword. If a row in which the city column does not exist in the data table, the search index considers that the city column does not exist.
  - The type of the city column in the search index is a basic type such as keyword. If a row in which the value of the city column is an empty array in the data table ("city" = "[]"), the search index considers that the city column does not exist.

API operations

You can call the Search or ParallelScan operation and set the query type to ExistsQuery to perform exists query.

Tablestore SDKs

You can use one of the following Tablestore SDKs to perform exists query:

- Tablestore SDK for Java: Exists query
- Tablestore SDK for Go: Exists query
- Tablestore SDK for Python: Exists query
- Tablestore SDK for PHP: Exists query

Parameters

Parameter	Description
fieldName	The name of the column.
query	The query type. Set the value to ExistsQuery.
getTotalCount	Specifies whether to return the total number of rows that meet the query conditions. The default value of this parameter is false, which indicates that the total number of rows that meet the query conditions is not returned. If you set this parameter to true, the query performance is compromised.
tableName	The name of the data table.

Parameter	Description
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each row that meets the query conditions. You can configure returnAll and columns for this parameter. The default value of returnAll is false, which indicates that not all columns are returned. In this case, you can use columns to specify the columns that you want to return. If you do not specify the columns that you want to return, only the primary key columns are returned. If you set returnAll to true, all columns are returned.

Examples

```
/**
 * Use ExistsQuery to query the rows in which the address column is not empty.
 * @param client
 */
private static void existQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    ExistsQuery existQuery = new ExistsQuery(); // Set the query type to ExistsQuery.
    existQuery.setFieldName("address");
    searchQuery.setQuery(existQuery);
    //searchQuery.setGetTotalCount(true); // Specify that the total number of rows that meet the query conditions is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    // You can use the columnsToGet parameter to specify the columns that you want to return or specify that all columns are returned. If you do not specify this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Specify the columns that you want to return.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that the total number of rows that meet the query conditions instead of the number of returned rows is displayed.
    System.out.println("Row: " + resp.getRows());
}
```

6.6.23. Collapse (distinct)

You can use the collapse feature to collapse the result set based on a specified column when the results of a query contain large amounts of data of a specific type. Data of the corresponding type is displayed only once in the query results to ensure diversity in the result types.

You can use the collapse feature to obtain distinct values based on collapsed columns in most scenarios. However, this feature is supported only for columns whose values are of the INTEGER, FLOATING-POINT or KEYWORD type. Only the first 10,000 sorted results are returned.

Usage notes

- If you use the collapse feature, you must implement pagination by specifying offset and limit instead of Token.
- If you aggregate and collapse a result set at the same time, the result set is aggregated before it is collapsed.
- If you collapse the query results, the total number of groups of the returned results is determined by the sum of the offset and limit values. A maximum of 10,000 groups can be returned.
- The total number of rows returned in the results indicates the number of returned rows before you use the collapse feature. After the result set is collapsed, the total number of the returned groups cannot be queried.

API operations

The API operation for the collapse (distinct) feature is Search, which is implemented by using the collapse parameter.

Usage

You can use the following Tablestore SDKs to implement the collapse (distinct) feature:

- Tablestore SDK for Java: Collapse (distinct)
- Tablestore SDK for Go: Collapse (distinct)
- Tablestore SDK for PHP: Collapse (distinct)

Parameters

Parameter	Description
query	The query type.
collapse	The collapse parameter, including the fieldName field. fieldName: the name of the column based on which the result set is collapsed. Only columns whose values are of the INTEGER, FLOATING-POINT or KEYWORD type are supported.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows without the data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without table data.

Parameter	Description
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when the total number of rows that match the query conditions is returned.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

```
private static void UseCollapse(SyncClient client){
    SearchQuery searchQuery = new SearchQuery(); // Specify the query conditions.
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("user_id");
    matchQuery.setText("00002");
    searchQuery.setQuery(matchQuery);
    Collapse collapse = new Collapse("product_name"); // Collapse the result set based on the product_name values.
    searchQuery.setCollapse(collapse);
    searchQuery.setOffset(1000);
    searchQuery.setLimit(20);
    //searchQuery.setGetTotalCount(true);//Set the total number of matched rows to return.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);// Specify the names of the table and the search index.
    // You can set the columnsToGet parameter to specify the columns to return or specify to return all columns. If you do not set this parameter, only the primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set returnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set columns to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse response = client.search(searchRequest);
    //System.out.println(response.getTotalCount()); // Display the total number of matched rows instead of the number of returned rows.
    //System.out.println(response.getRows().size()); // Display the number of the product name based on the product_name values.
    System.out.println(response.getRows()); // Display the product names based on the product_name values.
}
```

6.6.24. Aggregation

You can perform aggregation operations to obtain the minimum value, maximum value, sum, average, count and distinct count of rows, percentile statistics, and rows in each group. You can also perform aggregation operations to group results by field value, range, geographical location, filter, or histogram, and perform nested queries. You can perform multiple aggregation operations for complex queries.

Minimum value

The aggregation method that can be used to return the minimum value of a field. This method can be used in a similar manner as the SQL MIN function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 *  The price of each product is listed in the product table. Query the minimum price of the products that are produced in Zhejiang.
 *  SQL statement: SELECT min(column_price) FROM product where place_of_production = "Zhejiang".
 */
public void min(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production","Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead of specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.min("min_agg_1", "column_price").missing(100))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("min_agg_1").getValue());
}
```

Maximum value

The aggregation method that can be used to return the maximum value of a field. This method can be used in a similar manner as the SQL MAX function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The price of each product is listed in the product table. Query the maximum price of the products that are produced in Zhejiang.
 * SQL statement: SELECT max(column_price) FROM product where place_of_production = "Zhejiang".
 */
public void max(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead of specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.max("max_agg_1", "column_price").missing(0))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsMaxAggregationResult("max_agg_1").getValue());
}
```

Sum

The aggregation method that can be used to return the sum of all values for a numeric field. This method can be used in a similar manner as the SQL SUM function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The sales volume of each product is listed in the product table. Query the total numbe
r of the sold products that are produced in Zhejiang. The value of the missing parameter
is set to 10.
 * SQL statement: SELECT sum(column_price) FROM product where place_of_production = "Zhej
iang".
 */
public void sum(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead o
f specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.sum("sum_agg_1", "column_number").mis
sing(10))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("sum_agg_1"
).getValue());
}
```

Average

The aggregation method that can be used to return the average of all values for a numeric field. This method is used in a similar manner as the SQL AVG function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The sales volume of each product is listed in the product table. Query the average pri
ce of the products that are produced in Zhejiang.
 * SQL statement: SELECT avg(column_price) FROM product where place_of_production = "Zhej
iang".
 */
public void avg(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead o
f specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.avg("avg_agg_1", "column_number"))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsAvgAggregationResult("avg_agg_1"
).getValue());
}
```

Count

The aggregation method that can be used to return the total number of values for a specified field or the total number of rows in a table. This method can be used in a similar manner as the SQL COUNT function.

Note You can use one of the following methods to query the total number of rows in a table or the total number of rows that match the query conditions:

- Use the count feature of aggregation. Set the count parameter to * in the request.
- Use the query feature to obtain the number of rows that match the query conditions. Set the setGetTotalCount parameter to true in the query. Use MatchAllQuery to obtain the total number of rows in a table.

You can use the name of a column as the value of the count expression to query the number of rows that contain the column in a table. This method is suitable for scenarios that involve sparse columns.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.

Parameter	Description
fieldName	The name of the field that is used for the aggregation operation. Only the following data types are supported: LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT.

- Examples

```
/**
 * Punishment records of merchants are recorded in the merchant table. You can query the
number of merchants who are located in Zhejiang and for whom punishment records exist. If
no punishment records exist for a merchant, the field that corresponds to punishment reco
rds also does not exist for the merchant.
 * SQL statement: SELECT count(column_history) FROM product where place_of_production = "
Zhejiang".
 */
public void count(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place", "Zhejiang"))
                .limit(0)
                .addAggregation(AggregationBuilders.count("count_agg_1", "column_history"
))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsCountAggregationResult("count_ag
g_1").getValue());
}
```

Distinct count

The aggregation method that can be used to return the number of distinct values for a field. This method can be used in a similar manner as the SQL COUNT(DISTINCT) function.

Note The number of distinct values is an approximate number.

- If the total number of rows before the distinct count feature is used is less than 10,000, the calculated result is an exact value.
- If the total number of rows before the distinct count feature is used is greater than or equal to 100 million, the error rate is approximately 2%.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the following data types are supported: LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * Query the number of distinct provinces from which the products are produced.
 * SQL statement: SELECT count(distinct column_place) FROM product.
 */
public void distinctCount(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.distinctCount("dis_count_agg_1", "col
umn_place"))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
dis_count_agg_1").getValue());
}
```

Percentile statistics

A percentile value indicates the relative position of a value in a dataset. For example, when you collect statistics for the response time of each request during the routine O&M of your system, you must analyze the response time distribution by using percentiles such as p25, p50, p90, and p99.

> **Note** To improve the accuracy of the results, we recommend that you specify extreme percentile values such as p1 and p99. If you use extreme percentile values instead of other values such as p50, the returned results are more accurate.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the LONG and DOUBLE data types are supported.
percentiles	The percentiles such as p50, p90, and p99. You can specify one or more percentiles.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * Analyze the distribution of the response time of each request that is sent to the syst
em by using percentiles.
 */
public  void percentilesAgg(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addAggregation(AggregationBuilders.percentiles("percentilesAgg", "latenc
y")
                    .percentiles(Arrays.asList(25.0d, 50.0d, 99.0d))
                    .missing(1.0))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the analysis results.
    PercentilesAggregationResult percentilesAggregationResult = resp.getAggregationResult
s().getAsPercentilesAggregationResult(
        "percentilesAgg");
    for (PercentilesAggregationItem item : percentilesAggregationResult.getPercentilesAgg
regationItems()) {
        System.out.println("key: " + item.getKey() + " value:" + item.getValue().asDouble
());
    }
}
```

Group by field value

The aggregation method that can be used to group query results based on field values. The values that are the same are grouped together. The identical value of each group and the number of identical values in each group are returned.

> ? **Note** The calculated number may be different from the actual number if the number of values in a group is very large.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the following data types are supported: LONG, DOUBLE, BOOLEAN, and KEYWORD.
groupBySorter	The sorting rules for items in a group. By default, group items are sorted in descending order. If you configure multiple sorting rules, data is sorted based on the order in which the rules are configured. Supported parameters: ○ Sort by value in alphabetical order ○ Sort by value in reverse alphabetical order ○ Sort by row count in ascending order ○ Sort by row count in descending order ○ Sort the values that are obtained from sub-aggregation results in ascending order ○ Sort the values that are obtained from sub-aggregation results in descending order
size	The number of returned groups. Maximum value: 2000. If the number of groups exceeds 2,000, only the first 2,000 groups are returned.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results. ○ Scenario Query the number of products in each category, and the maximum and minimum product prices in each category. ○ Method Group query results by product category to obtain the number of products in each category. Then, perform two sub-aggregation operations to obtain the maximum and minimum product prices in each category. ○ Examples ■ Fruits: 5. The maximum price is 15. The minimum price is 3. ■ Toiletries: 10. The maximum price is 98. The minimum price is 1. ■ Electronic devices: 3. The maximum price is 8,699. The minimum price is 2,300. ■ Other products: 15. The maximum price is 1,000. The minimum price is 80.

- Example 1:

```
/**
 * Query the number of products, and the maximum and minimum product prices in each categ
ory.
 * Example of returned results: Fruits: 5. The maximum price is 15, and the minimum price
is 3. Toiletries: 10. The maximum price is 98, and the minimum price is 1. Electronic dev
ices: 3. The maximum price is 8,699, and the minimum price is 2,300. Other products: 15.
The maximum price is 1,000, and the minimum price is 80.
 */
public void groupByField(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByField("name1", "column_type")
                    .addSubAggregation(AggregationBuilders.min("subName1", "column_price"
))
                    .addSubAggregation(AggregationBuilders.max("subName2", "column_price"
))
                )
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("
name1").getGroupByFieldResultItems()) {
        // Display values.
        System.out.println(item.getKey());
        // Display the number of rows.
        System.out.println(item.getRowCount());
        // Display the minimum prices.
        System.out.println(item.getSubAggregationResults().getAsMinAggregationResult("sub
Name1").getValue());
        // Display the maximum prices.
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("sub
Name2").getValue());
    }
}
```

- Example 2

```
    /**
     * Group results based on multiple fields.
     * Search indexes do not support the GROUP BY clause for multiple fields in SQL. You
can use nested GroupBy parameters to obtain the results that are the same as those obtain
ed by using the GROUP BY clause for multiple fields.
     * SQL statement: select a,d, sum(b),sum(c) from user group by a,d.
     */
    public void GroupByMultiField() {
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .returnAllColumns(true)   // You can set returnAllColumns to false and specif
y a value for addColumesToGet to have better query performance.
            //.addColumnsToGet("col_1","col_2")
            .searchQuery(SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())   // Specify query conditions. Query con
ditions can be used in the same manner as the WHERE clause in SQL. You can use QueryBuild
ers.bool() to perform nested queries.
                .addGroupBy(
                    GroupByBuilders
                        .groupByField("unique name_1", "field_a")
                        .size(20)
                        .addSubGroupBy(
                            GroupByBuilders
                                .groupByField("unique name_2", "field_d")
                                .size(20)
                                .addSubAggregation(AggregationBuilders.sum("unique name_3
", "field_b"))
                                .addSubAggregation(AggregationBuilders.sum("unique name_4
", "field_c"))
                        )
                )
                .build())
            .build();
        SearchResponse response = client.search(searchRequest);
        // Query rows that meet the specified conditions.
        List<Row> rows = response.getRows();
        // Obtain the aggregation results.
        GroupByFieldResult groupByFieldResult1 = response.getGroupByResults().getAsGroupB
yFieldResult("unique name_1");
        for (GroupByFieldResultItem resultItem : groupByFieldResult1.getGroupByFieldResul
tItems()) {
            System.out.println("field_a key:" + resultItem.getKey() + " Count:" + resultI
tem.getRowCount());
            // Obtain the sub-aggregation results.
            GroupByFieldResult subGroupByResult = resultItem.getSubGroupByResults().getAs
GroupByFieldResult("unique name_2");
            for (GroupByFieldResultItem item : subGroupByResult.getGroupByFieldResultItem
s()) {
                System.out.println("field_a " + resultItem.getKey() + " field_d key:" + i
tem.getKey() + " Count: " + item.getRowCount());
                double sumOf_field_b = item.getSubAggregationResults().getAsSumAggregatio
nResult("unique name_3").getValue();
                double sumOf_field_c = item.getSubAggregationResults().getAsSumAggregatio
nResult("unique name_4").getValue();
                System.out.println("sumOf_field_b:" + sumOf_field_b);
                System.out.println("sumOf_field_c:" + sumOf_field_c);
            }
        }
    }
```

- Example 3

```
    /**
     * Configure sorting rules for aggregation.
     * Method: Configure sorting rules by specifying GroupBySorter. If you configure multiple sorting rules, data is sorted based on the order in which the rules are configured. GroupBySorter supports sorting in ascending or descending order.
     * By default, data is sorted by row count in descending order. You can sort data by using GroupBySorter.rowCountSortInDesc().
     */
    public void groupByFieldWithSort(SyncClient client) {
        // Create a query statement.
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .searchQuery(
                SearchQuery.newBuilder()
                    .query(QueryBuilders.matchAll())
                    .limit(0)
                    .addGroupBy(GroupByBuilders
                        .groupByField("name1", "column_type")
                        //.addGroupBySorter(GroupBySorter.subAggSortInAsc("subName1")) // Sort data in ascending order based on the values that are obtained from sub-aggregation results.
                        .addGroupBySorter(GroupBySorter.groupKeySortInAsc())           // Sort data in ascending order based on the values that are obtained from aggregation results.
                        //.addGroupBySorter(GroupBySorter.rowCountSortInDesc())        // Sort data in descending order based on the number of rows that are obtained from the aggregation results.
                        .size(20)
                        .addSubAggregation(AggregationBuilders.min("subName1", "column_price"))
                        .addSubAggregation(AggregationBuilders.max("subName2", "column_price"))
                    )
                    .build())
            .build();
        // Execute the query statement.
        SearchResponse resp = client.search(searchRequest);
    }
```

Group by range

The aggregation method that can be used to group query results based on the value ranges of a field. Field values that are within a specified range are grouped together. The number of values in each range is returned.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.

Parameter	Description
fieldName	The name of the field that is used for the aggregation operation. Only the LONG and DOUBLE data types are supported.
range[double_from, double_to)	The value ranges for grouping. The value range can start from Double.MIN_VALUE and end with Double.MAX_VALUE.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results. For example, after you group query results by sales volume and by province, you can obtain the province that has the largest proportion of sales volume in a specified range. You must specify a value for GroupByField in GroupByRange to perform this query.

- Examples

```
/**
 * Group sales volumes based on ranges [0, 1000), [1000, 5000), and [5000, Double.MAX_VAL
UE) to obtain the sales volume in each range.
 */
public void groupByRange(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByRange("name1", "column_number")
                    .addRange(0, 1000)
                    .addRange(1000, 5000)
                    .addRange(5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    for (GroupByRangeResultItem item : resp.getGroupByResults().getAsGroupByRangeResult("
name1").getGroupByRangeResultItems()) {
        // Display the number of rows.
        System.out.println(item.getRowCount());
    }
}
```

Group by geographical location

The aggregation method that can be used to group query results based on geographical locations to a central point. Query results in distances that are within a specified range are grouped together. The number of items in each range is returned.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the GEOPOINT data type is supported.
origin(double lat, double lon)	The longitude and latitude of the central point. double lat specifies the latitude of the central point. double lon specifies the longitude of the central point.
range[double_from, double_to)	The distance ranges that are used for grouping. Unit: meters. The start value of the value range can be Double.MIN_VALUE and the end value can be Double.MAX_VALUE.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results.

- Examples

```
/**
 * Group users based on geographical locations to a Wanda Plaza to obtain the number of u
sers in each distance range. The distance ranges are [0, 1000), [1000, 5000), and [5000,
Double.MAX_VALUE). Unit: meters.
 */
public void groupByGeoDistance(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByGeoDistance("name1", "column_geo_point")
                    .origin(3.1, 6.5)
                    .addRange(0, 1000)
                    .addRange(1000, 5000)
                    .addRange(5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    for (GroupByGeoDistanceResultItem item : resp.getGroupByResults().getAsGroupByGeoDist
anceResult("name1").getGroupByGeoDistanceResultItems()) {
       // Display the number of rows.
        System.out.println(item.getRowCount());
    }
}
```

Group by filter

The aggregation method that can be used to filter the query results and group them together to obtain the number of results that match each filter. Results are returned in the order in which the filters are specified.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
filter	The filters that can be used for the query. Results are returned in the order in which the filters are specified.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results.

- Examples

```
/**
 * Specify the following filters to obtain the number of items that match each filter: Th
e sales volume exceeds 100, the place of origin is Zhejiang, and the description contains
Hangzhou.
 */
public void groupByFilter(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByFilter("name1")
                    .addFilter(QueryBuilders.range("number").greaterThanOrEqual(100))
                    .addFilter(QueryBuilders.term("place","Zhejiang"))
                    .addFilter(QueryBuilders.match("text","Hangzhou"))
                )
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results based on the order of filters.
    for (GroupByFilterResultItem item : resp.getGroupByResults().getAsGroupByFilterResult
("name1").getGroupByFilterResultItems()) {
        // Display the number of rows.
        System.out.println(item.getRowCount());
    }
}
```

Query by histogram

The aggregation method that can be used to group query results based on specific data intervals. Field values that are within the same range are grouped together. The value range of each group and the number of values in each group are returned.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
interval	The data interval that is used to obtain aggregation results.
fieldRange[min,max]	The range that is used together with the interval parameter to limit the number of groups. The value that is calculated by using the `(fieldRange.max-fieldRange.min)/interval` formula cannot exceed 2,000.

Parameter	Description
minDocCount	The minimum number of rows. If the number of rows in a group is less than the minimum number of rows, the aggregation results for the group are not returned.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * Collect statistics on the distribution of users by age group.
 */
public static void groupByHistogram(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addGroupBy(GroupByBuilders
                    .groupByHistogram("groupByHistogram", "age")
                    .interval(10)
                    .minDocCount(0L)
                    .addFieldRange(0, 99))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = ots.search(searchRequest);
    // Obtain the results that are returned when the aggregation operation is performed.
    GroupByHistogramResult results = resp.getGroupByResults().getAsGroupByHistogramResult
("groupByHistogram");
    for (GroupByHistogramItem item : results.getGroupByHistogramItems()) {
        System.out.println("key:" + item.getKey().asLong() + " value:" + item.getValue())
;
    }
}
```

Query the rows that are obtained from the results of an aggregation operation in each group

After you group query results, you can query the rows in each group. This method can be used in a similar manner as ANY_VALUE(field) in MySQL.

Note When you query the rows that are obtained from the results of an aggregation operation in each group, the returned results contain only the primary key information if the search index contains the NESTED, GEOPOINT, or ARRAY field. To obtain the required field, you must query the data table.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
limit	The maximum number of rows that can be returned for each group. By default, only one row of data is returned.
sort	The sorting method that is used to sort data in groups.
columnsToGet	The fields that you want to return. Only fields in search indexes are supported. ARRAY, GEOPOINT, and NESTED fields are not supported.

- Examples

```
/**
 * An activity application form of a school contains fields in which information such as
the names of students, classes, head teachers, and class presidents can be specified. You
can group students by class to view the application statistics and the property informati
on of each class.
 * SQL statement: select className, teacher, monitor, COUNT(*) as number from table GROUP
BY className.
 */
public void testTopRows(SyncClient client) {
    SearchRequest searchRequest = SearchRequest.newBuilder()
            .indexName("indexName")
            .tableName("tableName")
            .searchQuery(
                    SearchQuery.newBuilder()
                            .query(QueryBuilders.matchAll())
                            .limit(0)
                            .addGroupBy(GroupByBuilders.groupByField("groupName", "classN
ame")
                                    .size(5)
                                    .addSubAggregation(AggregationBuilders.topRows("topRo
wsName")
                                            .limit(1)
                                            .sort(new Sort(Arrays.asList(new FieldSort("t
eacher", SortOrder.DESC)))) // Sort rows by teacher in descending order.
                                    )
                            )
                            .build())
            .addColumnsToGet(Arrays.asList("teacher", "monitor"))
            .build();
    SearchResponse resp = client.search(searchRequest);
    List<GroupByFieldResultItem> items = resp.getGroupByResults().getAsGroupByFieldResult
("groupName").getGroupByFieldResultItems();
    for (GroupByFieldResultItem item : items) {
        String className = item.getKey();
        long number = item.getRowCount();
        List<Row> topRows = item.getSubAggregationResults().getAsTopRowsAggregationResult
("topRowsName").getRows();
        Row row = topRows.get(0);
        String teacher = row.getLatestColumn("teacher").getValue().asString();
        String monitor = row.getLatestColumn("monitor").getValue().asString();
    }
}
```

Nesting

GroupBy supports nesting. You can perform sub-aggregation operations by using GroupBy.

You can use nesting to perform sub-aggregation operations in a group. For example, you can perform nesting aggregation operations up to two levels.

- GroupBy + SubGroupBy: Items are grouped by province and by city to obtain data for each city in each province.
- GroupBy + SubAggregation: Items are grouped by province to obtain the maximum value of a metric for each province.

Note To ensure high performance of queries and perform GroupBy operations, you can specify only a small number of levels for nesting. For more information, see Search index limits.

Examples

```
/**
 * Perform nesting-based aggregation.
 * Two aggregations and one GroupByField attribute are specified in the outermost level. Tw
o sub-aggregations and one GroupByRange attribute are specified in GroupByField.
 */
public void subGroupBy(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .indexName("index_name")
        .tableName("table_name")
        .returnAllColumns(true)
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.match("textField", "hello"))
                .limit(10)
                .addAggregation(AggregationBuilders.min("name1", "fieldName1"))
                .addAggregation(AggregationBuilders.max("name2", "fieldName2"))
                .addGroupBy(GroupByBuilders
                    .groupByField("name3", "fieldName3")
                    .addSubAggregation(AggregationBuilders.max("subName1", "fieldName4"))
                    .addSubAggregation(AggregationBuilders.sum("subName2", "fieldName5"))
                    .addSubGroupBy(GroupByBuilders
                        .groupByRange("subName3", "fieldName6")
                        .addRange(12, 90)
                        .addRange(100, 900)
                    ))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the maximum and minimum values for the first level.
    AggregationResults aggResults = resp.getAggregationResults();
    System.out.println(aggResults.getAsMinAggregationResult("name1").getValue());
    System.out.println(aggResults.getAsMaxAggregationResult("name2").getValue());
    // Obtain the GroupByField results of the first level and the results of the aggregatio
ns that are nested in GroupByField.
    GroupByFieldResult results = resp.getGroupByResults().getAsGroupByFieldResult("someName
1");
    for (GroupByFieldResultItem item : results.getGroupByFieldResultItems()) {
        System.out.println("count:" + item.getRowCount());
        System.out.println("key:" + item.getKey());
        // Obtain the sub-aggregation results.
        // Display the maximum value that is obtained from the results of the sub-aggregati
on operation.
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("subNa
me1"));
        // Display the sum that is obtained from the results of the sub-aggregation operati
on.
```

```
        System.out.println(item.getSubAggregationResults().getAsSumAggregationResult("subName2"));
        // Display the GroupByRange values that are obtained from the results of the sub-aggregation operation.
        GroupByRangeResult subResults = resp.getGroupByResults().getAsGroupByRangeResult("subName3");
        for (GroupByRangeResultItem subItem : subResults.getGroupByRangeResultItems()) {
            System.out.println("count:" + subItem.getRowCount());
            System.out.println("key:" + subItem.getKey());
        }
    }
}
```

Multiple aggregations

You can perform multiple aggregation operations.

> **Note** If you perform multiple complex aggregation operations at the same time, a long period of time may be required.

- Example 1:

```
public void multipleAggregation(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the minimum value from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").getValue());
    // Obtain the sum from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").getValue());
    // Obtain the number of distinct values from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("name3").getValue());
}
```

- Example 2

```
public void multipleGroupBy(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .addGroupBy(GroupByBuilders.groupByField("name4", "type"))
                .addGroupBy(GroupByBuilders.groupByRange("name5", "long").addRange(1, 15)
)
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the minimum value from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").ge
tValue());
    // Obtain the sum from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").ge
tValue());
    // Obtain the number of distinct values from the results of the aggregation operation
.
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
name3").getValue());
    // Obtain the values of GroupByField from the results of the aggregation operation.
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("
name4").getGroupByFieldResultItems()) {
        // Display the keys.
        System.out.println(item.getKey());
        // Display the number of rows.
        System.out.println(item.getRowCount());
    }
    // Obtain the values of GroupByRange from the results of the aggregation operation.
    for (GroupByRangeResultItem item : resp.getGroupByResults().getAsGroupByRangeResult("
name5").getGroupByRangeResultItems()) {
        // Display the number of rows.
        System.out.println(item.getRowCount());
    }
}
```

6.6.25. Parallel scan

If you do not have requirements on the order of query results, you can use parallel scan to quickly obtain query results.

Background information

The search index feature allows you to call the Search API operation to query data, sort data in a specific order, and aggregate data.

In some cases, a faster query speed may be more important than the order of query results. For example, when you want to connect Tablestore to a cluster computing environment such as Spark or Presto, or you want to query a specified group of objects. To improve query speeds, Tablestore provides the ParallelScan API operation for the search index feature.

> **Note** Tablestore SDKs 5.6.0 and later versions support the parallel scan feature.

Compared with the Search operation, the ParallelScan operation supports all query features but does not provide analytics capabilities such as sorting and aggregation. This way, query speeds are improved by more than five times. You can call the ParallelScan operation to export hundreds of millions of data rows within a minute. The capability to export data can be horizontally scaled without upper limits.

The maximum number of rows that can be returned by each ParallelScan call is greater than the maximum number of rows that can be returned by each Search call. The Search operation returns up to 100 rows per call, whereas the ParallelScan operation returns up to 2,000 rows per call. The parallel scan feature allows you to use multiple threads to initiate requests in a session in parallel, which accelerates data export.

Scenarios

- If you want to sort or aggregate query results, or the query request is sent from an end user, use the Search operation.
- If you do not need to sort query results and want to quickly return all matched results, or the data is pulled by a computing environment such as Spark or Presto, use the ParallelScan operation.

Features

The differences between the ParallelScan operation and the Search operation lie in the following aspects:

- Stable results

  Parallel scan tasks are stateful. In a session, the result set of the scanned data is determined by the data status when the first request is initiated. If data is inserted or modified after the first request is sent, the result set is not affected.

- Sessions

  Parallel scan-related operations use sessions. The session ID can be used to determine the result set of scanned data. The following process describes how to obtain and use a session ID:

  i. Use the ComputeSplits operation to query the maximum number of parallel scan tasks and the current session ID.

  ii. Initiate multiple parallel scan requests to read data. You must specify the current session ID and the parallel scan task IDs in these requests.

  If the session ID is difficult to obtain, you can call the ParallelScan operation to initiate a request without a specified session ID. However, if you send a request without a specified session ID, there is a very low probability that the obtained result set contains duplicate data.

  Tablestore returns the OTSSessionExpired error code when network exceptions, thread exceptions, dynamic modifications on schemas, or index switchovers occur in the parallel scan process and data scans stop. In these cases, you must initiate another parallel scan task to scan data again.

- Maximum number of parallel scan tasks in a single request

  The maximum number of parallel scan tasks in a single request supported by the ParallelScan operation is determined by the return value of the ComputeSplits request. A larger volume of data requires more parallel scan tasks in a session.

  A single request is specified by one query statement. For example, if you use the Search operation to query results in which the value of city is Hangzhou, all data that matches this condition is returned in the result. However, if you use the ParallelScan operation and the number of parallel scan tasks in a session is 2, each ParallelScan request returns half of the results. The complete result set consists of the two parallel result sets.

- Maximum number of rows that can be returned by each ParallelScan call

  The default value of limit is 2000. The maximum value of limit is 2000. If you enter a value greater than 2000, the performance is not affected.

- Cost

  ParallelScan requests consume fewer resources and are offered at a lower price. To export large amounts of data, we recommend that you use the ParallelScan operation.

- Columns to return

  Only indexed columns can be returned from search indexes. You can set the ReturnType parameter to RETURN_ALL_INDEX or RETURN_SPECIFIED, but not to RETURN_ALL.

  The ParallelScan operation can return only values of the ARRAY and GEOPOINT columns. However, the return values are formatted and may be different from the values that are written to the data table. For example, if you write [1,2, 3, 4] to an ARRAY column, the ParallelScan operation returns [1,2,3,4] as the value. If you write `10,50` to a GEOPOINT column, the ParallelScan operation returns `10.0,50.0` as the value.

- Limits

  The maximum number of parallel scan tasks is 10. You can adjust this limit based on your business requirements. Parallel tasks that have the same session ID and the same ScanQuery parameter value are considered one task. A parallel scan task starts from the time when you send the first ParallelScan request, and ends when all data is scanned or the token expires.

API operations

You can call the following API operations to use the parallel scan feature:

- ComputeSplits: You can call this operation to query the maximum number of parallel scan tasks for a single ParallelScan request.
- ParallelScan: You can call this operation to export data.

Parameters

Parameter	Description
tableName	The name of the data table.
indexName	The name of the search index.

Parameter		Description
scanQuery	query	The query statement for the search index. The operation supports term query, fuzzy query, range query, geo query, and nested query, which are similar to those of the Search operation.
	limit	The maximum number of rows that can be returned by each ParallelScan call.
	maxParallel	The maximum number of parallel scan tasks per request. The maximum number of parallel scan tasks per request varies based on the data volume. A larger volume of data requires more parallel scan tasks per request. You can use the ComputeSplits operation to query the maximum number of parallel scan tasks per request.
	currentParallelId	The ID of the parallel scan task in the request. Valid values: [0, Value of maxParallel)
	token	The token that is used to paginate query results. The results of the ParallelScan request contain the token for the next page. You can use the token to retrieve the next page.
	aliveTime	The validity period of the current parallel scan task. This validity period is also the validity period of the token. Unit: seconds. Default value: 60. We recommend that you use the default value. If the next request is not initiated within the validity period, more data cannot be queried. The validity time of the token is refreshed each time you send a request. **Note** The server uses the asynchronous method to process expired tasks. The current task does not expire within the validity period. However, Tablestore does not guarantee that the task expires after the validity period.
columnsToGet		You can use parallel scan to scan data only in search indexes. To use parallel scan for a search index, you must set store to true when you create the search index.
sessionId		The session ID of the parallel scan task. You can call the ComputeSplits operation to create a session and query the maximum number of parallel scan tasks that are supported by the parallel scan request.

Examples

The following code provides examples on how to scan data by using a single thread or by using multiple threads at the same time:

- Scan data by using a single thread

  When you use parallel scan, the code for a request that uses a single thread is simpler than the code for a request that uses multiple threads. The currentParallelId and maxParallel parameters are not required for a request that uses a single thread. The ParallelScan request that uses a single thread provides higher throughput than the Search request. However, the ParallelScan request that uses a single thread provides lower throughput than the ParallelScan request that uses multiple threads. For more information about how to scan data by using multiple threads at the same time, see the "Scan data by using multiple threads" section.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ParallelScanResponse;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.SearchRequest.ColumnsToGet;
import com.alicloud.openservices.tablestore.model.search.query.MatchAllQuery;
import com.alicloud.openservices.tablestore.model.search.query.Query;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
public class Test {
    public static List<Row> scanQuery(final SyncClient client) {
        String tableName = "<TableName>";
        String indexName = "<IndexName>";
        // Query the session ID and the maximum number of parallel scan tasks supported b
y the request.
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsR
equest);
        byte[] sessionId = computeSplitsResponse.getSessionId();
        int splitsSize = computeSplitsResponse.getSplitsSize();
        /*
         * Create a parallel scan request.
         */
        ParallelScanRequest parallelScanRequest = new ParallelScanRequest();
        parallelScanRequest.setTableName(tableName);
        parallelScanRequest.setIndexName(indexName);
        ScanQuery scanQuery = new ScanQuery();
        // This query determines the range of the data to scan. You can create a nested a
nd complex query.
        Query query = new MatchAllQuery();
        scanQuery.setQuery(query);
        // Specify the maximum number of rows that can be returned by each ParallelScan c
all.
```

```
        scanQuery.setLimit(2000);
        parallelScanRequest.setScanQuery(scanQuery);
        ColumnsToGet columnsToGet = new ColumnsToGet();
        columnsToGet.setColumns(Arrays.asList("col_1", "col_2"));
        parallelScanRequest.setColumnsToGet(columnsToGet);
        parallelScanRequest.setSessionId(sessionId);
        /*
         * Use builder to create a parallel scan request that has the same features as th
e preceding request.
         */
        ParallelScanRequest parallelScanRequestByBuilder = ParallelScanRequest.newBuilder
()
            .tableName(tableName)
            .indexName(indexName)
            .scanQuery(ScanQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(2000)
                .build())
            .addColumnsToGet("col_1", "col_2")
            .sessionId(sessionId)
            .build();
        List<Row> result = new ArrayList<>();
        /*
         * Use the native API operation to scan data.
         */
        {
            ParallelScanResponse parallelScanResponse = client.parallelScan(parallelScanR
equest);
            // Query the token of ScanQuery for the next request.
            byte[] nextToken = parallelScanResponse.getNextToken();
            // Obtain the data.
            List<Row> rows = parallelScanResponse.getRows();
            result.addAll(rows);
            while (nextToken != null) {
                // Specify the token.
                parallelScanRequest.getScanQuery().setToken(nextToken);
                // Continue to scan the data.
                parallelScanResponse = client.parallelScan(parallelScanRequest);
                // Obtain the data.
                rows = parallelScanResponse.getRows();
                result.addAll(rows);
                nextToken = parallelScanResponse.getNextToken();
            }
        }
        /*
         * Recommended method.
         * Use an iterator to scan all matched data. This method has the same query speed
but is easier to use compared with the previous method.
         */
        {
            RowIterator iterator = client.createParallelScanIterator(parallelScanRequestB
yBuilder);
            while (iterator.hasNext()) {
                Row row = iterator.next();
                result.add(row);
                // Obtain the specific values.
                String col_1 = row.getLatestColumn("col_1").getValue().asString();
                long col_2 = row.getLatestColumn("col_2").getValue().asLong();
            }
        }
        /*
         * If the operation fails, retry the operation. If the caller of this function has a retry mechanism or if you do not want to retry the failed operation, you can ignore this part.
         * To ensure availability, we recommend that you start a new parallel scan task when exceptions occur.
         * The following exceptions may occur when you send a ParallelScan request:
         * 1. A session exception occurs on the server side. The error code is OTSSessionExpired.
         * 2. An exception such as a network exception occurs on the client side.
         */
        try {
            // Execute the processing logic.
            {
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequestByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    // Process rows of data. If you have sufficient memory resources, you can add the rows to a list.
                    result.add(row);
                }
            }
        } catch (Exception ex) {
            // Retry the processing logic.
            {
                result.clear();
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequestByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    // Process rows of data. If you have sufficient memory resources, you can add the rows to a list.
                    result.add(row);
                }
            }
        }
        return result;
    }
}
```

- Scan data by using multiple threads

```
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Semaphore;
import java.util.concurrent.atomic.AtomicLong;
public class Test {
    public static void scanQueryWithMultiThread(final SyncClient client, String tableName, String indexName) throws InterruptedException {
        // Query the number of CPU cores on the client.
        final int cpuProcessors = Runtime.getRuntime().availableProcessors();
        // Specify the number of parallel threads for the client. We recommend that you specify the number of CPU cores on the client as the number of parallel threads for the client to prevent impact on the query performance.
        final Semaphore semaphore = new Semaphore(cpuProcessors);
        // Query the session ID and the maximum number of parallel scan tasks supported by the request.
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsRequest);
        final byte[] sessionId = computeSplitsResponse.getSessionId();
        final int maxParallel = computeSplitsResponse.getSplitsSize();
        // Create an AtomicLong object if you need to obtain the row count for your business.
        AtomicLong rowCount = new AtomicLong(0);
        /*
         * If you want to perform multithreading by using a function, you can build an internal class to inherit the threads.
         * You can also build an external class to organize the code.
         */
        final class ThreadForScanQuery extends Thread {
            private final int currentParallelId;
            private ThreadForScanQuery(int currentParallelId) {
                this.currentParallelId = currentParallelId;
                this.setName("ThreadForScanQuery:" + maxParallel + "-" + currentParallelId);  // Specify the thread name.
            }
            @Override
            public void run() {
                System.out.println("start thread:" + this.getName());
                try {
                    // Execute the processing logic.
                    {
                        ParallelScanRequest parallelScanRequest = ParallelScanRequest.newBuilder()
                                .tableName(tableName)
                                .indexName(indexName)
                                .scanQuery(ScanQuery.newBuilder()
                                        .query(QueryBuilders.range("col_long").lessThan(10_0000)) // Specify the data to query.
                                        .limit(2000)
                                        .currentParallelId(currentParallelId)
                                        .maxParallel(maxParallel)
                                        .build())
                                .addColumnsToGet("col_long", "col_keyword", "col_bool")
// Specify the fields to return from the search index. To return all fields from the sear
ch index, set returnAllColumnsFromIndex to true.
                                //.returnAllColumnsFromIndex(true)
                                .sessionId(sessionId)
                                .build();
                        // Use an iterator to obtain all the data.
                        RowIterator ltr = client.createParallelScanIterator(parallelScanR
equest);
                        long count = 0;
                        while (ltr.hasNext()) {
                            Row row = ltr.next();
                            // Add a custom processing logic. The following sample code s
hows how to add a custom processing logic to count the number of rows:
                            count++;
                        }
                        rowCount.addAndGet(count);
                        System.out.println("thread[" + this.getName() + "] finished. this
thread get rows:" + count);
                    }
                } catch (Exception ex) {
                    // If exceptions occur, you can retry the processing logic.
                } finally {
                    semaphore.release();
                }
            }
        }
        // Simultaneously execute threads. Valid values of currentParallelId: [0, Value o
f maxParallel).
        List<ThreadForScanQuery> threadList = new ArrayList<ThreadForScanQuery>();
        for (int currentParallelId = 0; currentParallelId < maxParallel; currentParallelI
d++) {
            ThreadForScanQuery thread = new ThreadForScanQuery(currentParallelId);
            threadList.add(thread);
        }
        // Simultaneously initiate the threads.
        for (ThreadForScanQuery thread : threadList) {
            // Specify a value for semaphore to limit the number of threads that can be i
nitiated at the same time to prevent bottlenecks on the client.
            semaphore.acquire();
            thread.start();
        }
        // The main thread is blocked until all threads are complete.
        for (ThreadForScanQuery thread : threadList) {
            thread.join();
        }
        System.out.println("all thread finished! total rows:" + rowCount.get());
    }
}
```

6.7. Advanced features

6.7.1. Virtual columns

When you use the virtual column feature, you can modify the schema of a search index or create a search index to query new fields and data of new field types without modifying the storage schema and the data of Tablestore.

Purposes

The virtual column feature allows you to map a column in a data table to a virtual column in a search index when you create the search index. The type of the virtual column can be different from that of the column in the data table. This allows you to create a column without modifying the table schema and data. The new column can be used to accelerate queries or can be configured with different analyzers.

- You can configure different analyzers for a TEXT field.

  A single STRING column can be mapped to multiple TEXT columns of a search index. Different TEXT columns use different tokens to meet various business requirements.

- Accelerate queries

  You do not need to cleanse data or re-create a table schema. You need only to map required columns of a data table to the columns in a search index. The column types can be different between the data table and the search index. For example, you can convert the numeric type to the KEYWORD type to improve the performance of term query, and convert the STRING type to the numeric type to improve the performance of range query.

Precautions

- The following table describes the data type conversion between virtual columns and columns in data tables.

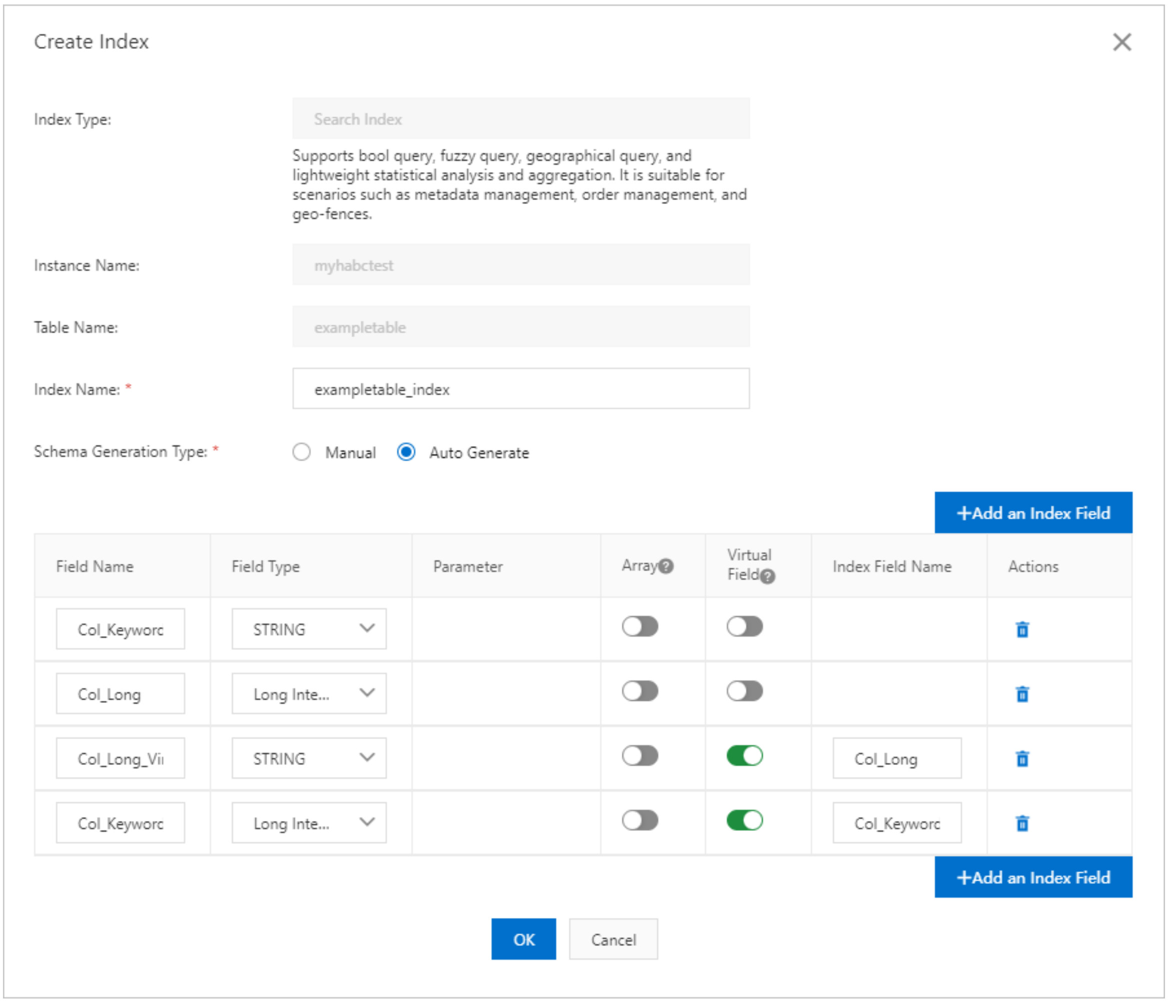
Field type of data tables	Field type of virtual columns
String	KEYWORD and KEYWORD ARRAY
String	TEXT and TEXT ARRAY
String	LONG and LONG ARRAY
String	DOUBLE and DOUBLE ARRAY
String	GEOPOINT and GEOPOINT ARRAY
Long	Keyword
Long	Text
Double	Keyword
Double	Text

- Virtual columns can be used only in query statements and cannot be used in ColumnsToGet to return column values. To return column values, you can specify that the system returns the source columns of the virtual columns.

Use the virtual column feature in the Tablestore console

After you specify a field as a virtual column when you create a search index in the Tablestore console, you can use the virtual column to query data.

1. Log on to the Tablestore console.
2. On the **Overview** page, click the name of the required instance or click **Manage Instance** in the Actions column that corresponds to the instance.
3. In the **Tables** section of the **Instance Details** tab, click the name of the data table whose search index you want to view and then click the **Indexes** tab. You can also click **Indexes** in the Actions column that corresponds to the data table.
4. On the **Indexes** tab, click **Create Search Index**.
5. In the **Create Index** dialog box, specify virtual columns when you create a search index.



Create Index

Index Type: Search Index

Supports bool query, fuzzy query, geographical query, and lightweight statistical analysis and aggregation. It is suitable for scenarios such as metadata management, order management, and geo-fences.

Instance Name: myhabctest

Table Name: exampletable

Index Name: * exampletable_index

Schema Generation Type: * ○ Manual ● Auto Generate

+Add an Index Field

Field Name	Field Type	Parameter	Array	Virtual Field	Index Field Name	Actions
Col_Keyworc	STRING		○	○		🗑
Col_Long	Long Inte...		○	○		🗑
Col_Long_Vi	STRING		○	●	Col_Long	🗑
Col_Keyworc	Long Inte...		○	●	Col_Keyworc	🗑

+Add an Index Field

OK Cancel

i. The system generates a search index name. You can also set Index Name to a specific value.

ii. Set Schema Generation Type.

- If you set Schema Generation Type to **Manual**, enter the field names. Set supported data types for the field values. Specify whether to turn on Array.
- If you set Schema Generation Type to **Auto Generate**, the system automatically uses the primary key columns and attribute columns of the data table as the index fields. Set supported data types for the field values. Specify whether to turn on Array.

> Note The **Field Name** and **Field Type** values must be the same as those of the data table. For more information about the mapping of field types between the data table and the search index, see Data types of column values.

iii. Create a virtual column.

> Notice When you create a virtual column, the data table must contain the name of the source field and the data type of the source field must match that of the virtual column.

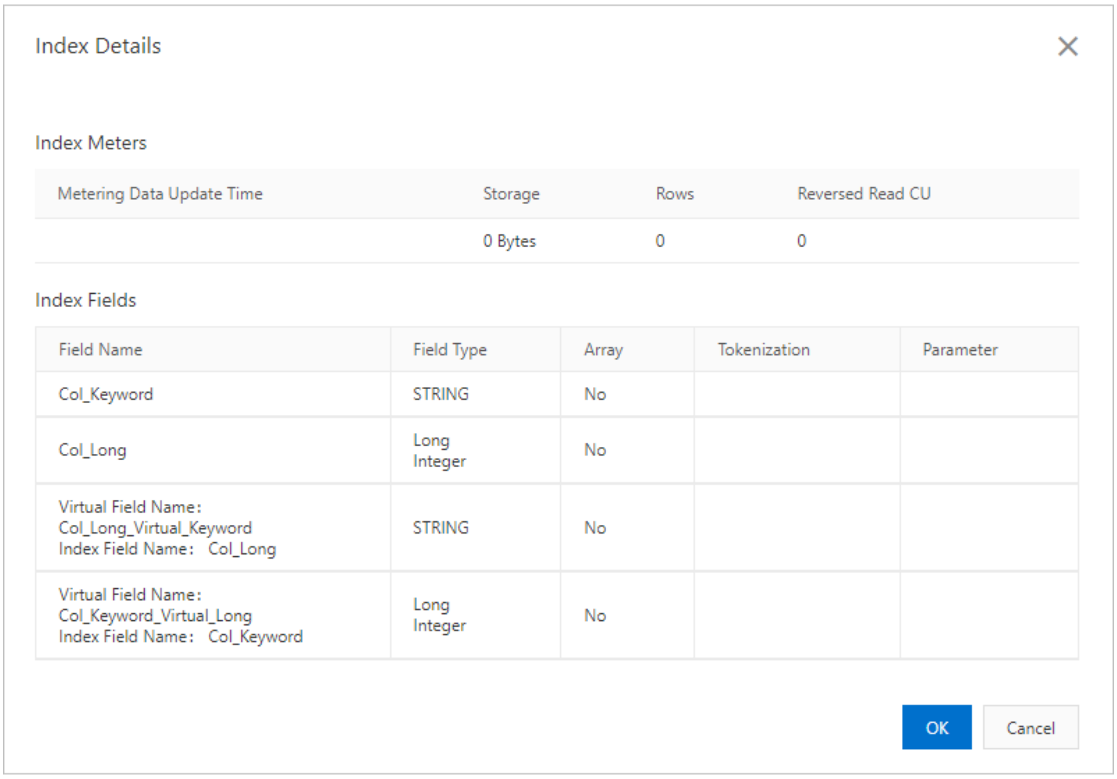
a. Click **Add an Index Field**.

b. Set **Field Name** and **Field Type**.

c. Turn on Virtual Column. Set **Index Field Name**.

iv. Click **OK**.

After the search index is created, click **Index Details** in the Actions column that corresponds to the search index. You can view the information about the search index, such as the metering information and index fields.



Index Details

Index Meters

Metering Data Update Time	Storage	Rows	Reversed Read CU
	0 Bytes	0	0

Index Fields

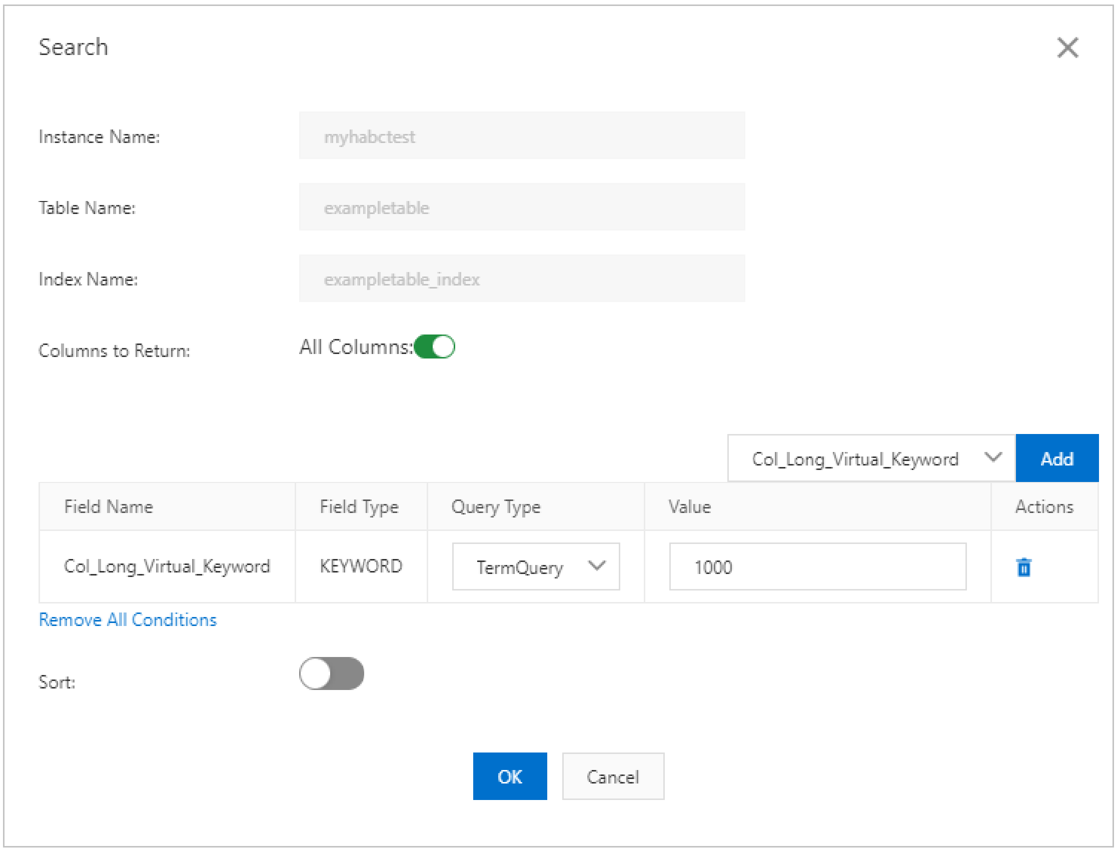
Field Name	Field Type	Array	Tokenization	Parameter
Col_Keyword	STRING	No		
Col_Long	Long Integer	No		
Virtual Field Name: Col_Long_Virtual_Keyword Index Field Name: Col_Long	STRING	No		
Virtual Field Name: Col_Keyword_Virtual_Long Index Field Name: Col_Keyword	Long Integer	No		

6. Use the virtual column to query data.

i. Click **Search** in the Actions column that corresponds to the search index.

i. Click Search in the Actions column that corresponds to the search index.

ii. In the **Search** dialog box, set filter conditions.



Search

Instance Name: myhabctest

Table Name: exampletable

Index Name: exampletable_index

Columns to Return: All Columns: on

Col_Long_Virtual_Keyword | Add

Field Name	Field Type	Query Type	Value	Actions
Col_Long_Virtual_Keyword	KEYWORD	TermQuery	1000	🗑

Remove All Conditions

Sort: off

OK Cancel

a. By default, the system returns all columns. To return specified attribute columns, turn off **All Columns**. Enter the attribute columns to return. Separate multiple attribute columns with commas (,).

b. Select index fields. Click **Add**. Set query methods and values for the fields.

c. By default, the sorting feature is disabled. To enable sorting, turn on **Sort** to sort query results based on the index fields. Add the index fields and configure sorting methods.

d. Click **OK**.

   Data that meets the filter conditions is displayed in the specified order on the **Data Editor** tab.

Use Tablestore SDKs to manage the virtual column feature

After you specify a field as a virtual column when you use a Tablestore SDK to create a search index, you can use the virtual column to query data.

1. Specify a virtual column when you create a search index.

   - Parameters

     For more information about parameters, see Create search indexes.

   - Examples

2. Use a virtual column to query data.

   Query the Col_Long_Virtual_Keyword column from a data table whose value can match "1000". Specify that the macthed rows and the total number of matched rows are returned.

```
private static void query(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermsQuery termsQuery = new TermsQuery(); // Set the query type to TermsQuery.
    termsQuery.setFieldName("Col_Long_Virtual_Keyword"); // Set the name of the field that you want to match.
    termsQuery.addTerm(ColumnValue.fromString("1000")); // Set the value that you want to match.
    searchQuery.setQuery(termsQuery);
    searchQuery.setGetTotalCount(true); // Specify that the total number of matched rows is returned.
    SearchRequest searchRequest = new SearchRequest("tableName", "indexName", searchQuery);
    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns without returning the virtual column.
    searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total number of matched columns but not returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

6.7.2. Dynamically modify schemas

You can dynamically modify the schema of a search index. For example, you can add, update, or delete index columns for the search index, and modify the routing keys of the search index.

Overview

Data tables of Tablestore are schema-free. However, search indexes have rigid schemas. When you create a search index, you must specify the columns you want to add to the search index. Then, you can query these columns when you use the search index to query data. To adapt to business changes and optimize performance, an increasing number of users need to modify the schemas of search indexes. Tablestore allows you to dynamically modify the schemas of search indexes in the following scenarios:

- Add index columns: You can add index columns if your business requires more columns for queries.
- Update index columns: Modify the analyzer of a TEXT field.
- Delete index columns: You may need to remove unnecessary columns added when you create a search index.
- Modify routing keys: You can specify routing keys to reduce read workloads and improve query efficiency.

The following process describes how to dynamically modify a schema with ease. The whole process does not affect business. You do not need to change business code.

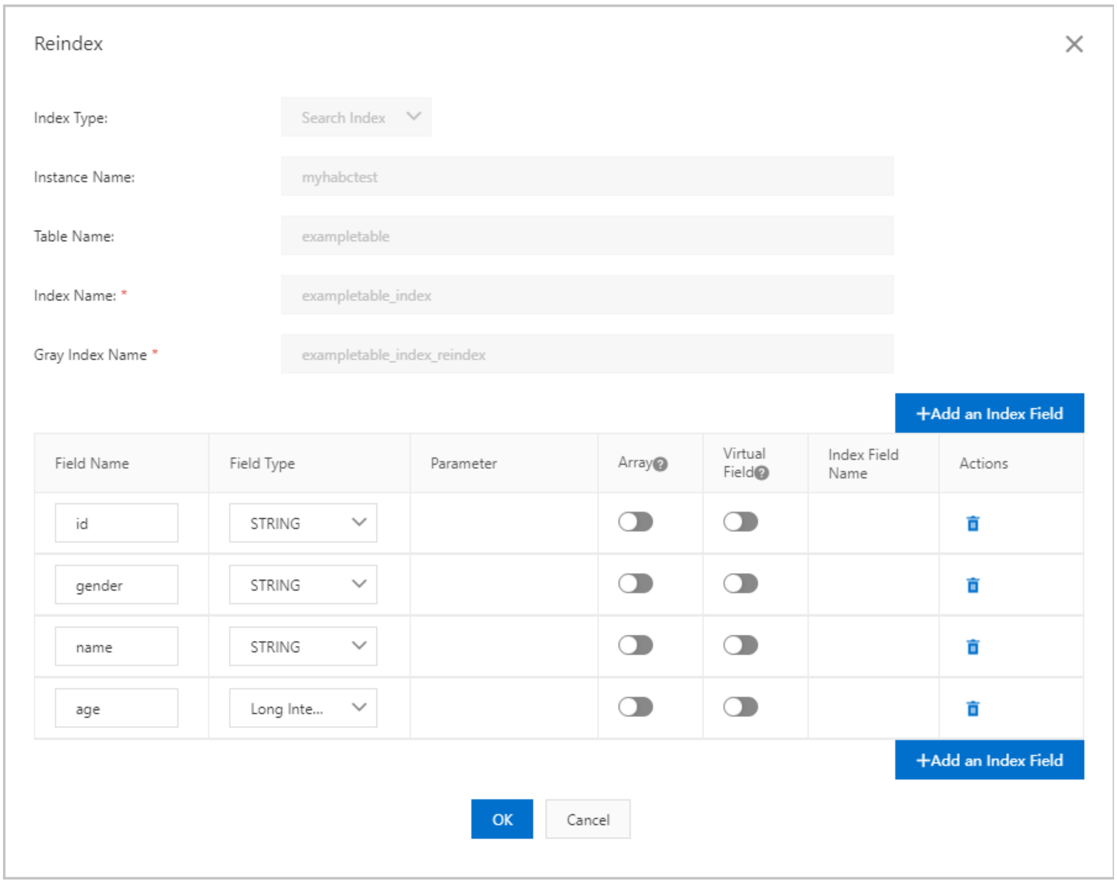
1. Create a grayscale index on a data table. Add, modify, or remove the schema of a search index.
2. Wait until the existing and incremental data of the data table is synchronized to the grayscale index and the synchronization progress is the same as that of the search index.
3. Use A/B testing to make sure that traffic is gradually directed to the grayscale index. Wait until all traffic is switched to the grayscale index.
4. After you verify that the grayscale index works normally, switch the schemas between the source

index and the grayscale index.

5. Delete the source index schema.

Procedure

1. Go to the **Indexes** tab.
   i. Log on to the Tablestore console.
   ii. On the **Overview** page, click the name of the required instance or click **Manage Instance** in the Actions column that corresponds to the instance.
   iii. In the **Tables** section of the **Instance Details** tab, click the name of the data table and then click the **Indexes** tab. You can also click **Indexes** in the Actions column that corresponds to the data table.
2. Create a grayscale index based on the source index.
   i. On the **Indexes** tab, click **Change Schema** in the Actions column that corresponds to the search index.
   ii. In the **Reindex** dialog box, add, modify, or delete index fields.



Reindex

Index Type: Search Index

Instance Name: myhabctest

Table Name: exampletable

Index Name: * exampletable_index

Gray Index Name * exampletable_index_reindex

+Add an Index Field

Field Name	Field Type	Parameter	Array	Virtual Field	Index Field Name	Actions
id	STRING		○	○		🗑
gender	STRING		○	○		🗑
name	STRING		○	○		🗑
age	Long Inte...		○	○		🗑

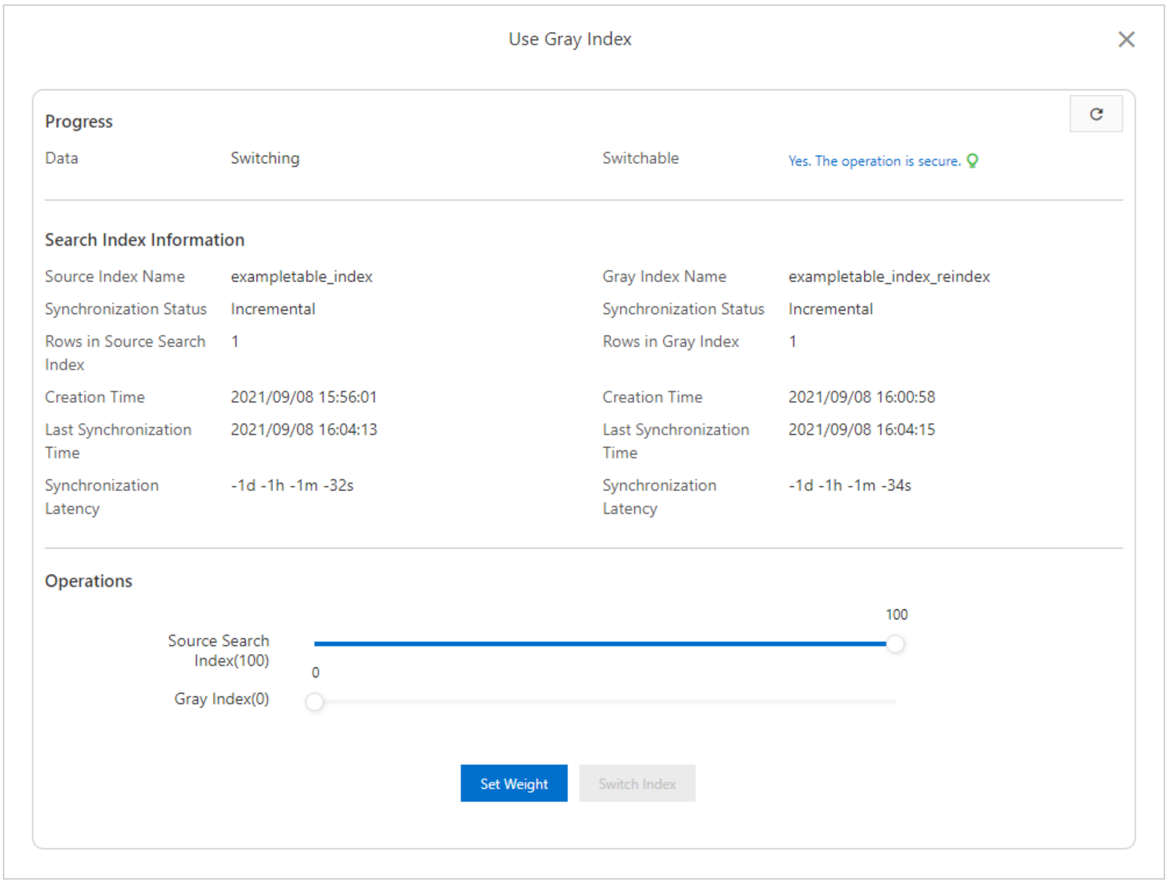
+Add an Index Field

OK Cancel

   iii. Click **OK**.
   iv. In the **Compare Schemas** message, compare the schema information between the source index and the grayscale index. After you confirm the information, click **OK**.
3. View the index synchronization information.

   The existing data synchronization and incremental data synchronization stages are required for the grayscale index. Before data is synchronized, the system displays **Yes, but the operation may**

cause security risks. In this case, you cannot perform the switchover. When the synchronization progress of the grayscale index is the same as that of the source index, the system displays **Yes. The operation is secure.** You can perform subsequent operations.



Use Gray Index

Progress

Data	Switching	Switchable	Yes. The operation is secure.

Search Index Information

Source Index Name	exampletable_index	Gray Index Name	exampletable_index_reindex
Synchronization Status	Incremental	Synchronization Status	Incremental
Rows in Source Search Index	1	Rows in Gray Index	1
Creation Time	2021/09/08 15:56:01	Creation Time	2021/09/08 16:00:58
Last Synchronization Time	2021/09/08 16:04:13	Last Synchronization Time	2021/09/08 16:04:15
Synchronization Latency	-1d -1h -1m -32s	Synchronization Latency	-1d -1h -1m -34s

Operations

i. Click the + icon in front of the source index or click the name of the source index.

The system displays the grayscale index of the source index.

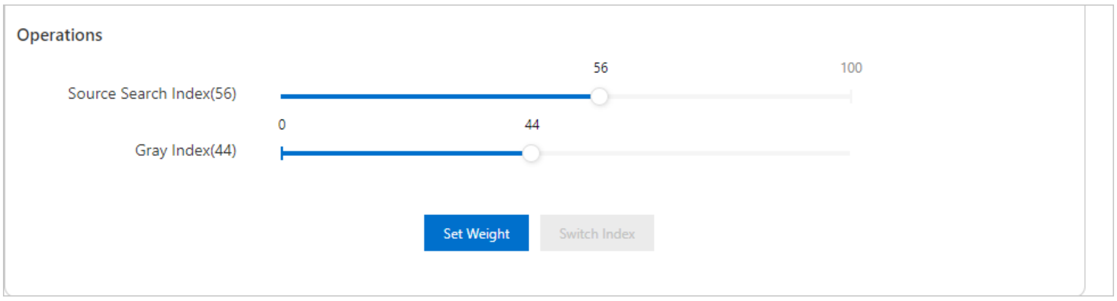
ii. Click **Use Gray Index** in the Actions column that corresponds to the grayscale index.

iii. In the **Use Gray Index** dialog box, view the synchronization information of the indexes.

4. After data is synchronized for the indexes, set weights to perform A/B testing.

A/B testing allows you to allocate traffic to the source index and the grayscale index based on proportions and verify the effects of changes to the schema. You can perform subsequent operations only when all traffic is directed to the grayscale index.

i. In the **Operations** section of the **Use Gray Index** dialog box, adjust the slider to control the weights for the source index and the grayscale index. Click **Set Weight**.



Operations

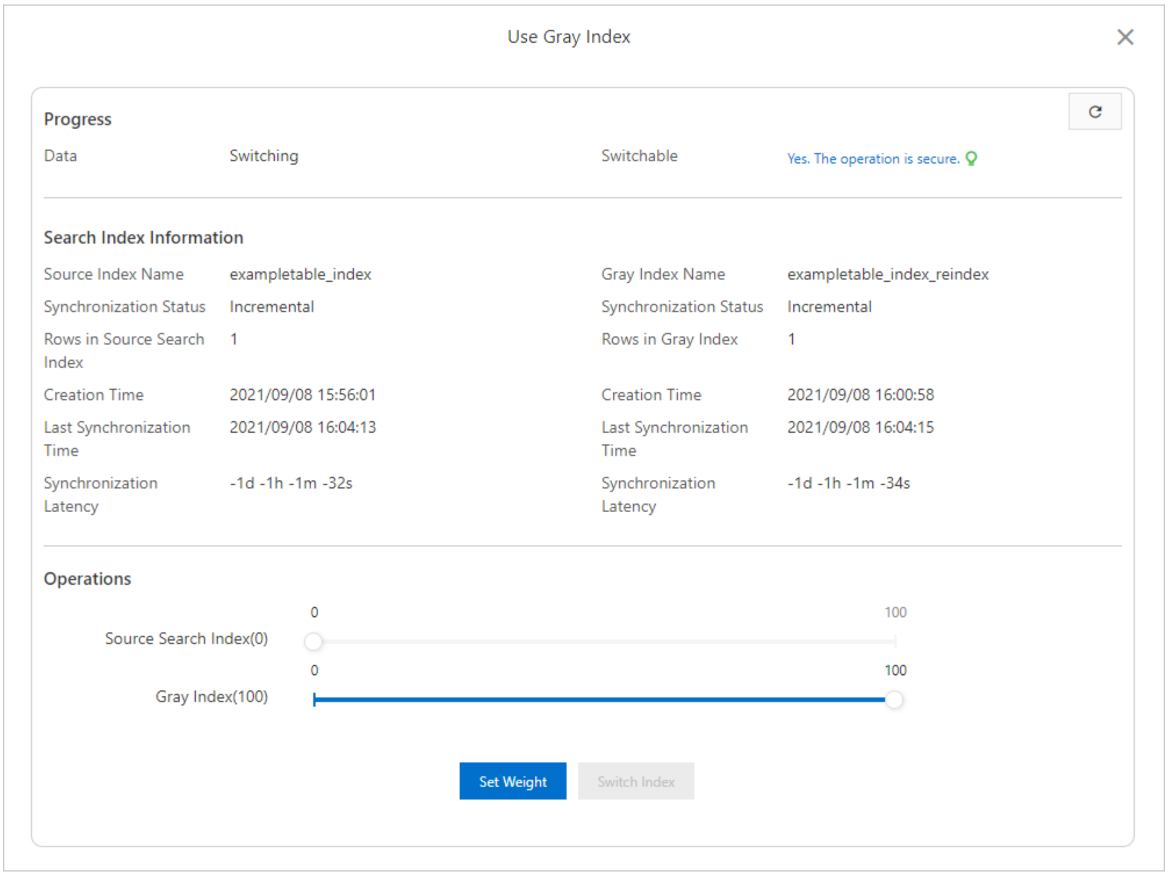
ii. In the **Set Weight** dialog box, view the weight data and the schema comparison information.

iii. After you confirm the information, click **Set Weight**.

iv. In the message that appears, click **OK**.

5. After all traffic for queries is directed to the grayscale index, switch the schemas between the source index and the grayscale index.

   After you switch the schemas, the name of the source index is associated with the new schema. The name of the grayscale index is associated with the source index schema. All traffic is directed to query the new schema associated with the source index name.



Use Gray Index

Progress

Data	Switching	Switchable	Yes. The operation is secure.

Search Index Information

Source Index Name	exampletable_index	Gray Index Name	exampletable_index_reindex
Synchronization Status	Incremental	Synchronization Status	Incremental
Rows in Source Search Index	1	Rows in Gray Index	1
Creation Time	2021/09/08 15:56:01	Creation Time	2021/09/08 16:00:58
Last Synchronization Time	2021/09/08 16:04:13	Last Synchronization Time	2021/09/08 16:04:15
Synchronization Latency	-1d -1h -1m -32s	Synchronization Latency	-1d -1h -1m -34s

Operations

Source Search Index(0)

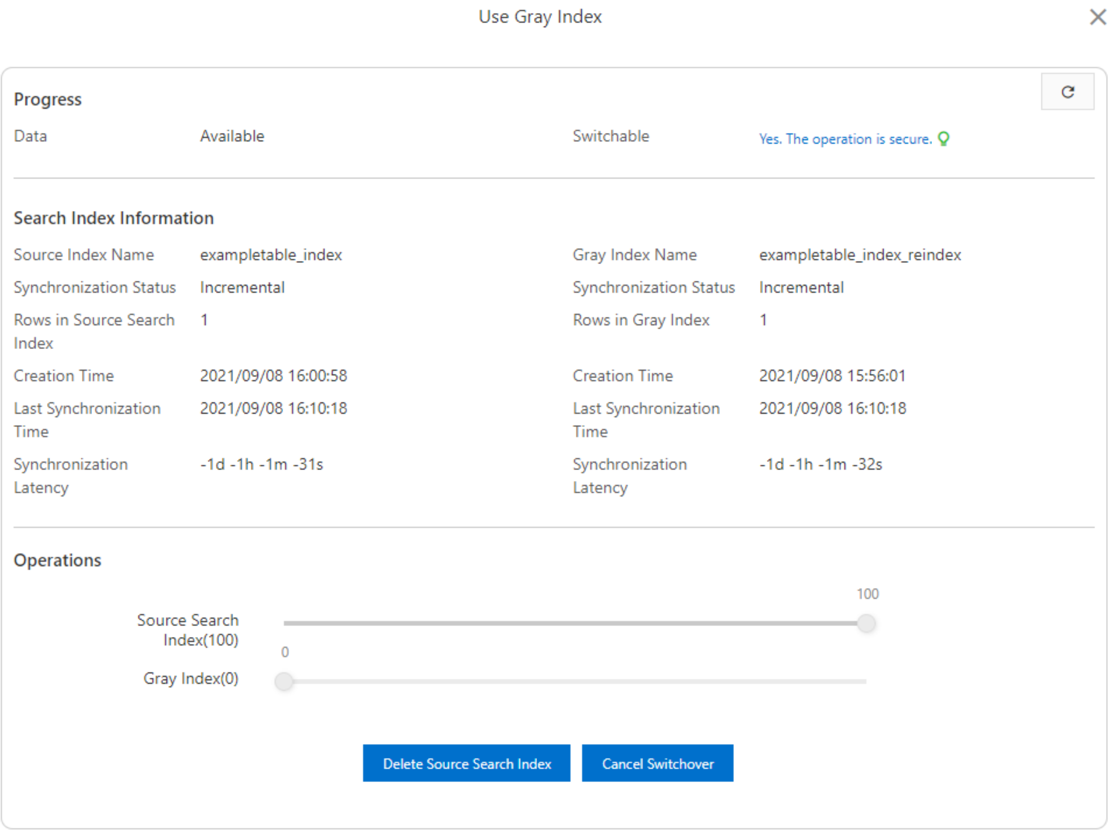
Gray Index(100)

i. In the **Operations** section of the **Use Gray Index** dialog box, click **Switch Index**.

ii. In the **Switch Index** dialog box, check the schema information of the source index and the grayscale index. Click **Confirm Switch**.

6. You can delete the source index schema after you verify that new schema is correct. To delete the source index schema, we recommend that you wait for a period of time such as one day.

   In the **Use Gray Index** dialog box, click **Delete Source Search Index**. You can delete the source index schema.



Use Gray Index

Progress

Data Available Switchable Yes. The operation is secure.

Search Index Information

Source Index Name	exampletable_index	Gray Index Name	exampletable_index_reindex
Synchronization Status	Incremental	Synchronization Status	Incremental
Rows in Source Search Index	1	Rows in Gray Index	1
Creation Time	2021/09/08 16:00:58	Creation Time	2021/09/08 15:56:01
Last Synchronization Time	2021/09/08 16:10:18	Last Synchronization Time	2021/09/08 16:10:18
Synchronization Latency	-1d -1h -1m -31s	Synchronization Latency	-1d -1h -1m -32s

Operations

Source Search Index(100) 100

Gray Index(0) 0

Security

To prevent incorrect operations, Tablestore provides the rollback mechanism and switchover notes to minimize the risks caused by modifying schemas.

- Rollback mechanism

  When you dynamically modify the schema of a search index, you can roll back the modification.

  - After you create a grayscale index, you can delete the grayscale index and create a grayscale index if the schema of the grayscale index does not meet your expectations.
  - When you perform A/B testing, you can configure weights to gradually direct traffic to the grayscale index. In this process, you can reset the weights anytime to direct traffic back to the source index if you find issues.
  - After you switch the schemas between the source index and the grayscale index, you can cancel the switchover anytime to switch back the schemas if you find issues. **Index switchover is the reverse of switchover cancellation.**

- Switchover notes

  If you switch traffic to a grayscale index when the synchronization progress of the grayscale index is slower than that of the source index, the data you query may not be the latest. At this time, Tablestore determines whether to switch indexes based on the synchronization status and the last synchronization time of the source index and the grayscale index.

  If the following situations exist, Tablestore determines that the indexes can be switched:

- The source index is in the full data synchronization stage. The grayscale index is in the full or incremental data synchronization stage. The synchronization progress of the grayscale index is the same as that of the source index.
- The source index and the grayscale index are in the incremental data synchronization stage. The last synchronization time of the source index is at most 60 seconds earlier than that of the grayscale index.

6.7.3. Fuzzy query

If you use `*word*` for a wildcard query (WildcardQuery), you can use fuzzy tokenization together with a match phrase query to obtain better query performance.

Background information

Fuzzy query is a common requirement in databases. For example, you can perform a fuzzy query to query file names and mobile numbers. To perform fuzzy queries in Tablestore, you can use the wildcard query feature of search indexes. The wildcard query feature is similar to the LIKE operator in MySQL. However, the wildcard query feature supports only up to 20 characters in the string that is used for a wildcard query, and the query performance decreases as the volume of data increases.

To resolve these issues, search indexes support fuzzy tokenization to ensure high performance in fuzzy queries. When you use fuzzy tokenization, Tablestore does not limit the length of the string that is used for a query. However, if the field value exceeds 1,024 characters in length, the system truncates the field value and performs tokenization only for the first 1,024 characters.

Scenarios

You can select a method that fits your scenario to perform a fuzzy query.

- If you use `*word*` for a wildcard query, you can use fuzzy tokenization to perform a fuzzy query. For example, if you use `"123"` to query mobile numbers that contain `123` at any position, you can use fuzzy tokenization to perform a fuzzy query.

  In this case, the fuzzy tokenization improves the query performance by more than 10 times than the wildcard query.

  For example, a data table contains a column named file_name, and the field type is Text and the tokenization method is fuzzy tokenization (Fuzzy_Analyzer) for the column in the search index. If you use the search index to query the rows whose file_name column value is `2021 woRK@Hangzhou`, you must perform a match phrase query (MatchPhraseQuery) and set the tokens to consecutive substrings for the query.

  - If the token for the query is `2021`, `20`, `21`, `work`, `WORK`, `@`, `Hang`, `zhou`, `Hang zhou`, or `@Hangzhou`, the rows whose file_name column value is `2021 woRK@Hangzhou` can match the token.
  - If the token for the query is `21work`, `2021Hangzhou`, `2120`, or `#Hangzhou`, the rows whose file_name column value is `2021 woRK@Hangzhou` cannot match the token.

- For other complex queries, you can use wildcard queries for fuzzy queries. For more information about the wildcard query, see Wildcard query.

Use fuzzy tokenization for a fuzzy query

To use fuzzy tokenization for a fuzzy query, perform the following steps:

1. Create a search index. When you create a search index, set the field type to Text and the tokenization method to fuzzy tokenization (Fuzzy Analyzer) for the specified column, and retain the default settings for other parameters.

   > **Note** If a search index exists, you can add a virtual column for the specified column by dynamically modifying the schema of the search index. Then, set the field type to Text and the tokenization method to fuzzy tokenization for the virtual column. For more information about the specific operations, see Dynamically modify schemas and Virtual columns.

2. Use the search index to query data. When you use the search index to query data, perform a match phrase query. For more information about the match phrase query, see Match phrase query.

Examples

The following test case shows how to use fuzzy tokenization to perform a fuzzy query:

```
package com.aliyun.tablestore.search.test;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.*;
import com.alicloud.openservices.tablestore.model.search.*;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
import com.aliyun.tablestore.search.common.Conf;
import com.aliyun.tablestore.search.common.TableStoreHelper;
import org.junit.Test;
import java.util.Arrays;
import java.util.Collections;
import static org.junit.Assert.assertEquals;
public class Test {
    private static final Conf conf = Conf.newInstance("src/test/resources/conf.json");
    private static final SyncClient ots = new SyncClient(conf.getEndpoint(), conf.getAccessId(), conf.getAccessKey(), conf.getInstanceName());
    private static final String tableName = "analysis_test";
    private static final String indexName = "analysis_test_index";
    @Test
    public void testFuzzyMatchPhrase() {
        // Delete the existing data table and index.
        TableStoreHelper.deleteTableAndIndex(ots, tableName);
        // Create a data table.
        TableStoreHelper.createTable(ots, tableName);
        // Define the schema of the data table.
        IndexSchema indexSchema = new IndexSchema();
        indexSchema.setFieldSchemas(Collections.singletonList(
                // Note: If you change the type of the name field for the query from Keyword to Text and set the tokenization method for the field, exceptions may occur in the query.

                // If you want to retain the fields of both the Keyword and Text types, see the example provided in the "Virtual columns" topic. If you use *abc* to match the name field, only the name field of the Text type is required. The name field of the Keyword type is not required.
                new FieldSchema("name", FieldType.TEXT).setAnalyzer(FieldSchema.Analyzer.Fuzzy)
        ));
        // Create a search index.
        TableStoreHelper.createIndex(ots, tableName, indexName, indexSchema);
        // Write a row of data to the data table.
        PrimaryKey primaryKey = PrimaryKeyBuilder.createPrimaryKeyBuilder()
                .addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("1"))
                .addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong(1))
                .addPrimaryKeyColumn("pk3", PrimaryKeyValue.fromBinary(new byte[]{1, 2, 3})
)
                .build();
        RowPutChange rowPutChange = new RowPutChange(tableName, primaryKey);
        // Add an attribute column to the data table.
        rowPutChange.addColumn("name", ColumnValue.fromString("TheLionKing1024x768P.mp4"));
        PutRowRequest request = new PutRowRequest(rowPutChange);
        ots.putRow(request);
        // Wait until the row of data is synchronized to the search index.
        TableStoreHelper.waitDataSync(ots, tableName, indexName, 1);
        // Use *abc* for the query.
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "The", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "TheLion", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "The Lion", 0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "TheLionKing102", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "TheLionKing1024", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "TheLionKing1024x", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "TheLionKing1024x7", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "TheLionKing1024x768P.mp4
", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "24x768P.mp4", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "24x76 8P.mp4", 0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "24x7 P.mp4", 0);
    }
    @Test
    // Use a virtual column.
    public void testFuzzyMatchPhraseWithVirtualField() {
        // Delete the existing data table and index.
        TableStoreHelper.deleteTableAndIndex(ots, tableName);
        // Create a data table.
        TableStoreHelper.createTable(ots, tableName);
        // Define the schema of the data table.
        IndexSchema indexSchema = new IndexSchema();
        indexSchema.setFieldSchemas(Arrays.asList(
                // Set the type of the name field to Keyword, which facilitates equivalent
queries.
                new FieldSchema("name", FieldType.KEYWORD).setIndex(true).setStore(true),
                // Create a virtual column named name_virtual_text and set the field type t
o Text and the tokenization method to Fuzzy for the virtual column. The data source of the
virtual column is the name field.
                new FieldSchema("name_virtual_text", FieldType.TEXT).setIndex(true).setAnal
yzer(FieldSchema.Analyzer.Fuzzy).setVirtualField(true).setSourceFieldName("name")
        ));
        // Create a search index.
        TableStoreHelper.createIndex(ots, tableName, indexName, indexSchema);
        // Write a row of data to the data table.
        PrimaryKey primaryKey = PrimaryKeyBuilder.createPrimaryKeyBuilder()
                .addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("1"))
                .addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong(1))
                .addPrimaryKeyColumn("pk3", PrimaryKeyValue.fromBinary(new byte[]{1, 2, 3})
)
                .build();
        RowPutChange rowPutChange = new RowPutChange(tableName, primaryKey);
        // Add an attribute column to the data table.
        rowPutChange.addColumn("name", ColumnValue.fromString("TheLionKing1024x768P.mp4"));
        PutRowRequest request = new PutRowRequest(rowPutChange);
        ots.putRow(request);
        // Wait until the row of data is synchronized to the search index.
        TableStoreHelper.waitDataSync(ots, tableName, indexName, 1);
        // Use *abc* for the query.
        // Note: The field for the query is name_virtual_text instead of name.
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "The", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "TheLion", 1
);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "The Lion",
0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "TheLionKing
102", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "TheLionKing
1024", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "TheLionKing
1024x", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "TheLionKing
1024x7", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "TheLionKing
1024x768P.mp4", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "24x768P.mp4
", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "24x76 8P.mp
4", 0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "24x7 P.mp4"
, 0);
    }
    // Perform a match phrase query.
    public static void assertMatchPhraseQuery(SyncClient ots, String tableName, String inde
xName, String fieldName, String searchContent, long exceptCount) {
        SearchRequest searchRequest = new SearchRequest();
        searchRequest.setTableName(tableName);
        searchRequest.setIndexName(indexName);
        SearchQuery searchQuery = new SearchQuery();
        // Perform a match phrase query to query data that matches the tokens.
        searchQuery.setQuery(QueryBuilders.matchPhrase(fieldName, searchContent).build());
        searchQuery.setLimit(0);
        // Specify that the total number of matched rows is returned. If you are not concer
ned about the total number of matched rows, set this parameter to false for better performa
nce.
        searchQuery.setGetTotalCount(true);
        searchRequest.setSearchQuery(searchQuery);
        SearchResponse response = ots.search(searchRequest);
        assertEquals(String.format("field:[%s], searchContent:[%s]", fieldName, searchConte
nt), exceptCount, response.getTotalCount());
    }
}
```

7.Secondary index

7.1. Overview

This topic describes the terms and the features of secondary indexes and the differences between secondary index types. This topic also describes the considerations when you use secondary indexes.

Background information

Secondary indexes allow you to create one or more index tables for a data table. Then, you can query data from the primary key columns of the index tables instead of the data table. This improves query performance.

Tablestore provides global secondary indexes and local secondary indexes to meet your requirements, such as strong query consistency.

Differences between secondary index types

Secondary indexes can be classified into global secondary indexes and local secondary indexes. You can use secondary indexes based on your requirements.

Name	Difference
Global secondary index	• Tablestore automatically synchronizes the data in indexed columns and primary key columns from a data table to an index table in asynchronous mode. In most cases, the synchronization is completed within milliseconds. • The first primary key column of the index table can be a primary key column or a predefined column of the data table.
Local secondary index	• Tablestore automatically synchronizes the data in indexed columns and primary key columns from a data table to an index table in synchronous mode. You can immediately query the data from the index table after the data is synchronized. • The first primary key column of the index table must be the first primary key column of the data table.

Terms

Term	Description
index table	The table created based on the indexed columns of a data table. The data in the index table is read-only.

Term	Description
predefined column	The non-primary column predefined when you create a data table. A predefined column is used as an attribute column of an index table. You can also specify the data type for the non-primary key column. Note Tablestore uses a schema-free model. You can write the data of different attribute columns to a row. You do not need to specify attribute columns for the table schema.
single-column index	The index that is created on a single column.
compound index	The index that is created on multiple columns. For example, a compound index can have indexed columns 1 and 2.
indexed attribute column	The predefined column mapped to an index table.
autocomplete	The feature that allows the system to automatically add the primary key columns that are not specified as indexed columns to an index table.

Features

- Single-column index and compound index

  You can create an index on one or more columns in a data table.

- Index synchronization

  Global secondary indexes and local secondary indexes synchronize data in different modes.

  - If you use global secondary indexes, Tablestore automatically synchronizes the data in indexed columns and primary key columns from a data table to an index table in asynchronous mode. In most cases, the synchronization is completed within milliseconds.
  - If you use local secondary indexes, Tablestore automatically synchronizes the data in indexed columns and primary key columns from a data table to an index table in synchronous mode. You can immediately query the data from the index table after the data is synchronized.

- Covered indexes

  Index tables can contain attribute columns. You can create predefined columns when you create a data table. Then, you can create an index table based on predefined columns and primary key columns of the data table. You can specify predefined columns as indexed attribute columns. You can also specify no indexed attribute columns.

  If you specify predefined columns of a data table as indexed attribute columns, you can query the values in the predefined columns from the index table. You do not need to query the data table.

- Index that contains existing data of a data table

  You can create an index table that contains the existing data of a data table.

- Sparse index

You can specify a predefined column of a data table as an indexed attribute column. If a row in the data table does not contain the predefined column but contains all indexed columns, an index is created on the row. However, an index cannot be created on a row if the row does not contain all indexed columns.

For example, a data table contains primary key columns PK0, PK1, and PK2, and predefined columns Defined0, Defined1, and Defined2. You create an index table that contains primary key columns PK0, Defined0, and Defined1, and an indexed attribute column Defined2.

- If a row of the data table contains Defined0 and Defined1 but does not contain Defined2, an index is created on the row.
- If a row of the data table contains Defined0 and Defined2 but does not contain Defined1, an index cannot be created on the row.

Limits

For more information, see Secondary index limits.

Considerations

When you configure indexed columns and attribute columns for an index table, take note of the following items:

- The system automatically adds the primary key columns that are not specified as indexed columns to an index table. When you scan data in an index table, you must specify the values of primary key columns. The values range from negative infinity to positive infinity.

  When you create an index table, you need only to specify the columns to be indexed. The system automatically adds all primary key columns of the data table to the index table. For example, a data table contains primary key columns PK0 and PK1 and a predefined column Defined0.

  If you use global secondary indexes, you can create an index on columns as needed.

  - If you create an index on Defined0, Tablestore generates the index table whose primary key columns are Defined0, PK0, and PK1.
  - If you create an index on Defined0 and PK1, Tablestore generates the index table whose primary key columns are Defined0, PK1, and PK0.
  - If you create an index on PK1, Tablestore generates the index table whose primary key columns are PK1 and PK0.

  If you use local secondary indexes, the first primary key column of an index table must be the same as the first primary key column of the data table.

  - If you create an index on PK0 and Defined0, Tablestore generates the index table whose primary key columns are PK0, Defined0, and PK1.
  - If you create an index on PK0, PK1, and Defined0, Tablestore generates an index table whose primary key columns are PK0, PK1, and Defined0.
  - If you create an index on PK0 and PK1, Tablestore generates the index table whose primary key columns are PK0 and PK1.

- You can specify predefined columns of a data table as indexed attribute columns based on your query modes and costs.

After you specify a predefined column of a data table as an indexed attribute column, you can query the value in the predefined column from the index table. You do not need to query the data table. However, this increases storage costs. If the predefined column of the data table is not specified as an indexed attribute column, you must query the column value from the data table.

- When you use global secondary indexes, specify a column of a data table as the first primary key column of an index table based on your requirements.
  - If the column whose value is time or a date is the first primary key column of an index table, the update speed of the index table may decrease. Therefore, we recommend that you do not specify this type of column as the first primary key column of an index table.

    We recommend that you hash the column whose value is time or a date and create an index on the hashed column. If you have such a requirement, use DingTalk to contact Tablestore technical support.
  - We recommend that you do not specify a column of low cardinality or a column that contains enumerated values as the first primary key column of an index table. For example, if you specify the gender column as the first primary key column of an index table, the horizontal scalability of the index table is limited. As a result, the write performance is compromised.

When you use an index table, take note of the following items:

- You must comply with the following rules when you write data to a data table that is associated with an index table. Otherwise, the data cannot be written to the data table.
  - You cannot customize the version number for the data that you write to an index table.
  - An index table cannot contain a row that has the same primary keys during a batch write operation.

Pricing

For more information, see Billable items of secondary indexes.

7.2. Scenarios

The secondary index feature allows you to create an index on the specified columns. Data in the generated index table is sorted by the specified indexed columns. All data written to the data table is automatically synchronized to the index table. If you only write data to the data table and then query the index table that is created on the data table, the query performance can be improved in most scenarios.

Global secondary index

You can use the global secondary index feature to perform various queries based on a data table of phone calls.

The following data table records the information about phone calls. When a phone call is complete, the information about the phone call is recorded in the data table.

- The CellNumber and StartTime columns act as the primary key columns of the data table. CellNumber indicates the caller, and StartTime the call start time.
- CalledNumber, Duration, and BaseStationNumber act as the predefined columns of the data table. CalledNumber indicates the number of a call recipient, Duration the duration of the call, and BaseStationNumber the base station number.

CellNumber	StartTime (UNIX timestamp)	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

You can create an index table based on the CalledNumber and BaseStationNumber columns to perform various queries. For the sample code that is used to create an index table, see Appendix.

You can use the global secondary index feature to meet the following query requirements:

- Query the rows where the value of CellNumber is 234567.

  Tablestore sorts rows in a data table based on their primary keys and provides the getRange operation. You can call the getRange operation to query the rows where the value of CellNumber is 234567. Set both the maximum and minimum values of CellNumber to 234567, and the minimum value of StartTime to 0 and the maximum value to INT_MAX. Then, scan the data table to obtain the expected data.

```
private static void getRangeFromMainTable(SyncClient client, long cellNumber){
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    // Construct the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.fromLo
ng(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLo
ng(0));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Construct the end primary key.
    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("The cell number " + strNum + "makes the following calls:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- Query the rows where the value of CalledNumber is 123456.

  Tablestore sorts rows in a data table based on their primary keys. Queries that involve predefined columns such as the CalledNumber column in this table are slow and inefficient. You can create an index table named IndexOnBeCalledNumber based on the CalledNumber column. CalledNumber becomes the primary key column of the index table. Therefore, you can call the getRange operation to scan the index table to obtain the expected data.

  The following table provides the information of IndexOnBeCalledNumber.

  Note Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. The primary key columns of the data table are added to the index table and act as the primary key columns in the index table. Therefore, the index table contains three primary key columns.

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

```
private static void getRangeFromIndexTable(SyncClient client, long cellNumber) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX0_NAME);
    // Construct the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.fromLo
ng(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MI
N);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MI
N);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Construct the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.fromLong
(cellNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
;
    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("The cell number" + strNum + "is called by the following numbers")
;
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- Query the rows where the value of BaseStationNumber is 002 and the value of StartTime is 1532574740.

  This query is similar to the query in the preceding example but uses two query conditions: BaseStationNumber and StartTime. You can create an index table named IndexOnBaseStation1 based on BaseStationNumber and StartTime. Then, you can query IndexOnBaseStation1 to obtain the expected data.

  The following table provides the information of IndexOnBaseStation1.

PK0	PK1	PK2
BaseStationNumber	StartTime	CellNumber

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

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX1_NAME);
    // Construct the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLong(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MIN);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Construct the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLong(baseStationNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX);
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strBaseStationNum = String.format("%d", baseStationNumber);
    String strStartTime = String.format("%d", startTime);
    System.out.println("All called numbers forwarded by the base station" + strBaseStationNum + "that start from" + strStartTime + "are listed:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

- Query the rows where the value of BaseStationNumber is 003 and the value of StartTime ranges from 1532574861 to 1532584054 and return only the values of the Duration column.

  This query must meet the conditions of BaseStationNumber and StartTime and return only the values of the Duration column. You can query IndexOnBaseStation1 used in the preceding example. Then, query the values of the Duration column from the data table.

```
private static void getRowFromIndexAndMainTable(SyncClient client,
                                                long baseStationNumber,
                                                long startTime,
                                                long endTime
                                                long endTime,
                                                String colName) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX1_NAME);
    // Construct the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLo
ng(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLo
ng(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MI
N);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Construct the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLong
(baseStationNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong
(endTime));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
;
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strBaseStationNum = String.format("%d", baseStationNumber);
    String strStartTime = 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" + strStartTime + "to" + strEndTime + "is listed:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn mainCalledNumber = curIndexPrimaryKey.getPrimaryKeyColumn(PR
IMARY_KEY_NAME_1);
            PrimaryKeyColumn callStartTime = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMA
RY_KEY_NAME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBuil
der();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, mainCalledNumber.g
etValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, callStartTime.getV
alue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); // Construct the primary
key for the data table.
            // Query the data table.
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, main
TablePK);
            criteria.addColumnsToGet(colName); // Read the Duration column values of the
data table.
            // Set MaxVersions to 1 to read the latest version of data.
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(mainTableRow);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

To make queries more efficient, you can create an index table named IndexOnBaseStation2 based on the BaseStationNumber and StartTime columns and specify the Duration column as an indexed attribute column. Then, query IndexOnBaseStation2 to obtain the expected data.

The following table provides the information of IndexOnBaseStation2.

PK0	PK1	PK2	Defined0
BaseStationNumber	StartTime	CellNumber	Duration
001	1532574644	123456	60
001	1532574714	234567	10
002	1532574795	345678	5
002	1532574861	345678	100
003	1532574734	234567	20
003	1532584054	456789	200

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime,
                                           long endTime,
                                           String colName) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX2_NAME);
    // Construct the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLo
ng(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLo
ng(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MI
N);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Construct the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLong
(baseStationNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong
(endTime));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
;
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    // Specify the name of the column to read.
    rangeRowQueryCriteria.addColumnsToGet(colName);
    rangeRowQueryCriteria.setMaxVersions(1);
    String strBaseStationNum = String.format("%d", baseStationNumber);
    String strStartTime = 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" + strStartTime + "to" + strEndTime + "is listed:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
...
```

If you do not specify Duration as an indexed attribute column, you must query the index table to obtain the primary key columns of the data table and then use the obtained primary key columns to query the data table. If you specify Duration as an indexed attribute column, Duration is stored in both the data table and the index table, which increases storage costs.

- Query the total, average, maximum, and minimum call duration of all phone calls forwarded by the 003 base station. The phone call start time ranges from 1532574861 to 1532584054.

  In this query, you want to obtain the statistics for the duration of all phone calls instead of the duration of each call queried in the preceding example. You can obtain results by using the same method as in the preceding example. Then, you can perform calculations on the Duration column to obtain the required results. You can also use Data Lake Analytics (DLA) to obtain the required results by executing SQL statements without client-side calculations.

> **Note** For more information about how to use DLA, see OLAP on Tablestore: Serverless SQL big data analysis based on Data Lake Analytics. DLA supports most MySQL syntax and can be used to perform complicated calculations that are applicable to your business.

Local secondary index

You can use the local secondary index feature to perform various queries on a data table of phone calls.

The following data table records the information about phone calls. When a phone call is complete, the information about the phone call is recorded in the data table.

- The CellNumber and StartTime columns act as the primary key columns of the data table. CellNumber indicates the caller, and StartTime the call start time.
- CalledNumber, Duration, and BaseStationNumber act as the predefined columns of the data table. CalledNumber indicates the number of a call recipient, Duration the duration of the call, and BaseStationNumber the base station number.

CellNumber	StartTime (UNIX timestamp)	CalledNumber	Duration	BaseStationNumber
123456	1532574644	654321	60	1
123456	1532574704	236789	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
456789	1532585054	123456	200	3
456789	1532586054	234567	200	3
456789	1532587054	123456	200	3

You can use the local secondary index feature to create an index table named LocalIndexOnBeCalledNumber based on CalledNumber. Then, you can query the records of phone calls between called numbers and caller numbers.

The following table provides the information of LocalIndexOnBeCalledNumber.

> **Note** Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. The primary key columns of the data table are added to the index table and act as the primary key columns in the index table. Therefore, the index table contains three primary key columns.

PK0	Defined0	PK1	Defined1	Defined2
CellNumber	CalledNumber	StartTime (UNIX timestamp)	Duration	BaseStationNumber
123456	236789	1532574704	60	1
123456	654321	1532574644	60	1
234567	123456	1532574734	20	3
234567	765432	1532574714	10	1
345678	123456	1532574795	5	2
345678	123456	1532574861	100	2
456789	123456	1532585054	200	3
456789	123456	1532587054	200	3
456789	234567	1532586054	200	3
456789	345678	1532584054	200	3

To query the records of phone calls between caller number 456789 and called number 123456, you need only to specify the following values when you call the getRange operation: Set the maximum and minimum values of CellNumber in the PK0 column to 456789. Set the maximum and minimum values of CalledNumber in the Defined0 column to 123456. Set the minimum value of StartTime in the PK1 column to 0 and the maximum value to INT_MAX. Then, scan the data table to obtain the expected data. The following code provides an example on how to list the records of phone calls:

```
private static void getRangeFromMainTable(SyncClient client, long cellNumber, long calledNu
mber){
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    // Construct the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_0, PrimaryKeyValue.fromLong
(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_0, PrimaryKeyValue.fromLong
(calledNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.fromLong
(0));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Construct the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_0, PrimaryKeyValue.fromLong(c
ellNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_0, PrimaryKeyValue.fromLong(c
alledNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX);
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strNum = String.format("%d", cellNumber);
  String strCalledNum = String.format("%d", calledNumber);
    System.out.println("All records of phone calls between the caller number" + strNum + "a
nd the called number" +strCalledNum+ "are listed:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQue
ryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextStart
PrimaryKey());
        } else {
            break;
        }
    }
}
```

7.3. Operations

This topic describes the operations you can call to use secondary indexes.

The following table describes the operations you can call to use global secondary indexes.

Operation	Description

Operation	Description
CreateIndex	Creates one or more index tables for a data table. Note • You can specify whether to include the existing data of a data table in an index table that you create by calling the CreateIndex operation. • You can create one or more index tables when you create a data table by calling the CreateTable operation.
	Reads a row of data from an index table.
GetRange	Reads data within a specified range from an index table.
DropIndex	Deletes the specified index table from a data table. Note Before you call the DeleteTable operation to delete a data table, you must first delete the index tables that are created on the data table. Otherwise, you cannot delete the data table.

7.4. Use SDK

7.4.1. Global secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement global secondary index:

- Tablestore SDK for Java: Global secondary index
- Tablestore SDK for Go: Global secondary index
- Tablestore SDK for Python: Global secondary index
- Tablestore SDK for Node.js: Global secondary index
- .Tablestore SDK for .NET: Global secondary index
- Tablestore SDK for PHP: Global secondary index

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table on an existing data table.

Note You can create one or more index tables when you create a data table by calling the CreateTable operation. For more information, see Create data tables.

- Parameters

Parameter	Description
mainTableName	The name of the data table.

Parameter	Description
indexMeta	The schema information of the index table. The schema information contains the following items: ○ indexName: the name of the index table. ○ primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ○ definedColumns: the indexed attribute columns. The attribute columns are a combination of predefined columns of the data table. ○ indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to be indexed and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. ○ indexUpdateMode: the update mode of the index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ■ If indexUpdateMode is not specified or is set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ■ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the index update mode to IUM_SYNC_INDEX, which indicates the synchronous update mode.

Parameter	Description
includeBaseData	Specifies whether to include the existing data of the data table in the index table. If the last parameter includeBaseData in CreateIndexRequest is set to true, the existing data of the data table is included in the index table. If includeBaseData is set to false, the existing data is excluded.

- Examples

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX2_NAME); // Specify the name of the index table.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // Specify DEFINED_COL_NAME_1 as the first primary key column of the index table.
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2); // Specify PRIMARY_KEY_NAME_2 as the second primary key column of the index table.
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); // Specify DEFINED_COL_NAME_2 as an attribute column of the index table.
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); // Create the index table on the data table. Specify that the index table includes the existing data of the data table.
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); // Create the index table on the data table. Specify that the index table excludes the existing data from the data table.
    /**You can set the IncludeBaseData parameter to true to synchronize existing data from the data table to the index table after the index table is created. Then, you can query all data from the index table.
       The amount of time required to synchronize data to the index table is determined by the amount of data in the data table.
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); // Create the index table.
}
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  When you call the GetRow operation to read data from an index table, take note of the following items:

  - You must set tableName to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from the index table

  For more information, see Multi-row operations.

  - Parameters

    When you call the GetRange operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

  - Examples

    If an index table contains the columns to return, you can query the required data from the index table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("Results returned from the index table:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

If an index table does not contain the columns to return, you must query the required data from the data table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
```

```
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MAX); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_NAME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, ke2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); // Specify primary key columns for the data table based on the primary key columns of the index table.
            // Query the data table.
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, mainTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME3); // Specify DEFINED_COL_NAME3 to read the DEFINED_COL_NAME3 attribute column from the data table.
            // Set MaxVersions to 1 to read the latest version of data.
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
indexName	The name of the index table.

- Examples

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME, INDEX_NAME); // Specify the name of the index table that you want to delete and the name of the corresponding data table.
    client.deleteIndex(request); // Delete the index table.
}
```

7.4.2. Local secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement the local secondary index feature:

- Tablestore SDK for Java: Local secondary index
- Tablestore SDK for Go: Local secondary index
- Tablestore SDK for Python: Local secondary index
- Tablestore SDK for Node.js: Local secondary index

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table on an existing data table.

> **Note** You can create one or more index tables when you create a data table by calling the CreateTable operation. For more information, see Create data tables.

- Parameter

Parameter	Description
mainTableName	The name of the data table.

Parameter	Description
indexMeta	The schema information of the index table. The schema information contains the following items: ◦ indexName: the name of the index table. ◦ primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ◦ definedColumns: the indexed attribute columns. The attribute columns are a combination of predefined columns of the data table. ◦ indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ▪ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to be indexed and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ▪ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. ◦ indexUpdateMode: the update mode of the index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ▪ If indexUpdateMode is not specified or is set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ▪ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the index update mode to IUM_SYNC_INDEX, which indicates the synchronous update mode.

Parameter	Description
includeBaseData	Specifies whether to include the existing data of the data table in the index table. If the last parameter includeBaseData in CreateIndexRequest is set to true, the existing data of the data table is included in the index table. If includeBaseData is set to false, the existing data is excluded.

- Examples

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); // Specify the name of the index table.
    indexMeta.setIndexType(IT_LOCAL_INDEX);     // Set the index type to IT_LOCAL_INDEX, which specifies a local secondary index.
    indexMeta.setIndexUpdateMode(IUM_SYNC_INDEX);  // Set the index update type to IUM_SYNC_INDEX, which specifies the synchronous update mode. If the index update type is set to IT_LOCAL_INDEX, the index update mode must be set to IUM_SYNC_INDEX.
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1); // Add a primary key column to the index table. The first primary key column of the index table must be the same as the first primary key column of the data table.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_2); // Set DEFINED_COL_NAME_2 to the second primary key column of the index table.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // Set DEFINED_COL_NAME_1 to the third primary key column of the index table.
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); // Create the index table for the data table. Specify that the index table contains the existing data from the data table.
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); // Create the index table for the data table. Specify that the index table does not contain the existing data from the data table.
    /**You can set the IncludeBaseData parameter to true to synchronize existing data from the data table to the index table after the index table is created. Then, you can query all data from the index table.
       The amount of time required to synchronize data to the index table varies based on the amount of data in the data table.
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); // Create the index table.
}
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  When you call the GetRow operation to read data from an index table, take note of the following items:

  - You must set tableName to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from the index table

  For more information, see Multi-row operations.

  - Parameters

    When you call the GetRange operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

  - Examples

    If an index table contains the columns to return, you can query the required data from the index table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME); // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MAX); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("Results returned from the index table:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

If the index table does not contain the columns to return, you must query the required data from the data table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME); // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBu
ilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, pk2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); // Specify primary key
columns for the data table based on the primary key columns of the index table.
            // Query the data table.
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, ma
inTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME3); // Read the DEFINED_COL_NAME3
attribute column from the data table.
            // Set MaxVersions to 1 to read the latest version of data.
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
indexName	The name of the index table.

- Examples

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME, INDEX_NAME); // Specify the name of the index table that you want to delete and the name of the corresponding data table.
    client.deleteIndex(request); // Delete the index table.
}
```

7.5. Appendix

This topic describes the appendix of global secondary index.

The following code provides an example on how to create a base table and an index table:

```
private static final String TABLE_NAME = "CallRecordTable";
    private static final String INDEX0_NAME = "IndexOnBeCalledNumber";
    private static final String INDEX1_NAME = "IndexOnBaseStation1";
    private static final String INDEX2_NAME = "IndexOnBaseStation2";
    private static final String PRIMARY_KEY_NAME_1 = "CellNumber";
    private static final String PRIMARY_KEY_NAME_2 = "StartTime";
    private static final String DEFINED_COL_NAME_1 = "CalledNumber";
    private static final String DEFINED_COL_NAME_2 = "Duration";
    private static final String DEFINED_COL_NAME_3 = "BaseStationNumber";
    private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType.INTEGER));
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_3, DefinedColumnType.INTEGER));
        int timeToLive = -1; // Specify the validity period of data in seconds. A value of -1 indicates that the data never expires. You must set the timeToLive value to -1 when the base table has one or more index tables.
        int maxVersions = 1; // Specify the maximum number of versions. You must set the maxVersions value to 1 when the base table has one or more index tables.
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
        IndexMeta indexMeta0 = new IndexMeta(INDEX0_NAME);
        indexMeta0.addPrimaryKeyColumn(DEFINED_COL_NAME_1);
        indexMetas.add(indexMeta0);
        IndexMeta indexMeta1 = new IndexMeta(INDEX1_NAME);
        indexMeta1.addPrimaryKeyColumn(DEFINED_COL_NAME_3);
        indexMeta1.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMetas.add(indexMeta1);
        IndexMeta indexMeta2 = new IndexMeta(INDEX2_NAME);
        indexMeta2.addPrimaryKeyColumn(DEFINED_COL_NAME_3);
        indexMeta2.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMeta2.addDefinedColumn(DEFINED_COL_NAME_2);
        indexMetas.add(indexMeta2);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMetas);
        client.createTable(request);
    }
```

8.SQL query

8.1. Overview

You can use the SQL query feature to perform complex queries and analytics on data in Tablestore in an efficient manner. The SQL query feature provides a unified access interface for multiple data engines.

Background information

Tablestore is a storage service that is developed by Alibaba Cloud. Tablestore can store large volumes of structured data by using multiple models, and supports fast data query and analytics. Empowered by the distributed storage and index-based search engine, Tablestore can store petabytes (PBs) of data while delivering tens of millions transactions per second (TPS) at ultra-low latency (milliseconds). You can use Tablestore to store and query large volumes of data in a convenient manner.

Tablestore provides cloud-native SQL engine capabilities in addition to the conventional storage of NoSQL structured data. SQL queries are compatible with MySQL query syntax and provide basic SQL Data Definition Language (DDL) statements to create tables. This way, you can migrate your business to Tablestore and access Tablestore by executing SQL statements. For existing data tables, you can execute the CREATE TABLE statement to create mapping tables for the existing data tables. Then, you can use SQL statements to access the data in the existing data tables.

The SQL query feature allows you to use search indexes to quickly query data that meets query conditions. When you use the SQL query feature, the system automatically selects an appropriate method to accelerate SQL access based on the SQL statements.

The SQL query feature is suitable for scenarios in which you want to access a large amount of data online. When you use the SQL query feature, the access latency ranges from milliseconds to minutes. The SQL query feature supports point queries based on the primary keys of data tables (GetRow operation), term queries based on search indexes (TermQuery), and queries based on the aggregation capability of search indexes, such as queries on the number of rows that meet a specified condition and the sum of values in a column.

Terms

The use of SQL involves many concepts in conventional databases. The following table describes some concepts in conventional databases and the mappings between the concepts and Tablestore concepts.

Term	Description
Database	A database is a data repository that organizes, stores, and manages data based on data structures. A database can contain one or more tables. Databases are mapped to Tablestore instances.
Table	A table consists of rows and columns. Tables in databases are mapped to Tablestore tables.
Index	Indexes in databases are mapped to secondary indexes or search indexes in Tablestore.

Features

- SQL features
  - Allows you to initiate a single request by executing a single SQL statement.
  - Supports basic DDL statements, including CREATE TABLE and DESCRIBE. For more information, see Create mapping tables for tables and Query the information about a table.
  - Supports Data Query Language (DQL) statements, including SELECT. For more information, see Query data.
  - Supports basic database administration statements, including SHOW TABLES and SHOW INDEX. For more information, see List table names and Query the index information about a table.
- Character set and collation
  - Character set: UTF-8
  - Collation: binary collation
- Operators

  Supports SQL operators such as arithmetic operators, relational operators, and logical operators. For more information, see SQL operators.
- Mapping tables for existing tables

  You can execute the CREATE TABLE statement to create mapping tables for existing tables. When you create a mapping table, make sure that the primary keys in the mapping table are the same as the primary keys in the existing table, and the attribute columns in the mapping table have the same types as the attribute columns and predefined columns in the existing table. For more information about data type mappings, see Data type mappings.

Usage notes

When you use the SQL query feature, the transaction feature is not supported.

Limits

For more information, see SQL limits.

Case conventions in SQL

The Tablestore SQL engine follows common SQL conventions and is not case-sensitive to column names. For example, the `SELECT Aa FROM exampletable;` statement is equivalent to the `SELECT aa FROM exampletable;` statement.

The column names in Tablestore are case-sensitive. When SQL is used, the column names in Tablestore are converted into lowercase letters for matching. For example, if you want to perform operations on the Aa column in a Tablestore table, you can use AA, aa, aA, or Aa in SQL. Therefore, the column names in Tablestore cannot be AA, aa, aA, and Aa at the same time.

Reserved words and keywords

Tablestore uses keywords in SQL statements as reserved words. If you need to use keywords to name tables or columns, add the `` `' `` symbol to escape the keywords. Keywords are not case-sensitive.

For more information about the reserved words and keywords, see Reserved words and keywords.

Comparison between SQL and search index

Search index		SQL function/statement
Term query		Equal (=)
Range query		Greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), and BETWEEN ... AND ...
Boolean query	MustQueries	AND
	MustNotQueries	!=
	ShouldQueries	OR
Sorting and pagination	FieldSort	ORDER BY
	SetLimit	LIMIT
Aggregation	Minimum value	MIN()
	Maximum value	MAX()
	Sum	SUM()
	Average value	AVG()
	Count	COUNT()
	Distinct count	COUNT(DISTINCT)
	Query the rows that are obtained from the results of an aggregation operation in each group	ANY_VALUE()
	Group by field value	GROUP BY

Billing

For more information, see Billable items of SQL query.

8.2. SQL features

This topic describes the SQL statements that are supported by the Tablestore SQL engine.

> **Notice** Starting on May 26, 2022, the SQL query feature is no longer provided free of charge. If an error occurs when you use the SQL query feature, submit a ticket.

SQL statement	Description	Supported
	Creates a mapping table for an existing data table.	Yes

SQL statement CREATE TABLE	Description	Supported
	Creates a mapping table for an existing search index.	Yes
ALTER TABLE	Adds, modifies, or removes columns.	Available soon
SHOW TABLES	Lists the names of tables in the current database.	Yes
SELECT	Queries data.	Yes
DROP MAPPING TABLE	Deletes a mapping table.	Yes
DROP TABLE	Deletes a data table.	Available soon
CREATE INDEX	Creates an index.	No
SHOW INDEX	Queries information about an index.	Yes
INSERT	Writes data.	No
SELECT JOIN	Performs a data query that joins multiple tables.	No

8.3. Data type mappings

This topic describes the mappings between the data types of fields in SQL, data tables, and search indexes. Before you use SQL, make sure that the field data types in SQL match the field data types in data tables.

Note

- If you use VARBINARY and VARCHAR as the data types of primary key columns in SQL, we recommend that you set the maximum length of the primary key column values to 1,024 by using VARBINARY(1024) and VARCHAR(1024).
- Both BIGINT in SQL and Integer in data tables are 64-bit integer data types.
- You can use only BIGINT, VARBINARY, and VARCHAR as the types of primary key column values in SQL.

Field data type in SQL	Field data type in data tables	Field data types in search indexes
BIGINT	Integer	Integer
• VARBINARY (primary key) • MEDIUMBLOB (attribute column)	Binary	Binary
• VARCHAR (primary key) • MEDIUMTEXT (attribute column)	String	Keyword
		Text

Field data type in SQL	Field data type in data tables	Field data types in search indexes
DOUBLE	Double	Double
BOOL	Boolean	Boolean

8.4. Use the Tablestore console

Tablestore allows you to use the SQL query feature to quickly query data. After you create a mapping table in the console, you can execute the SELECT statement to quickly query the required data.

Prerequisites

- If you want to use a RAM user to perform operations, make sure that a RAM user is created and all SQL operation permissions are granted to the RAM user. You can configure `"Action": "ots:SQL*"` in the custom policy to grant all SQL operation permissions to the RAM user. For more information, see Grant permissions to a RAM user.
- A data table is created.

Precautions

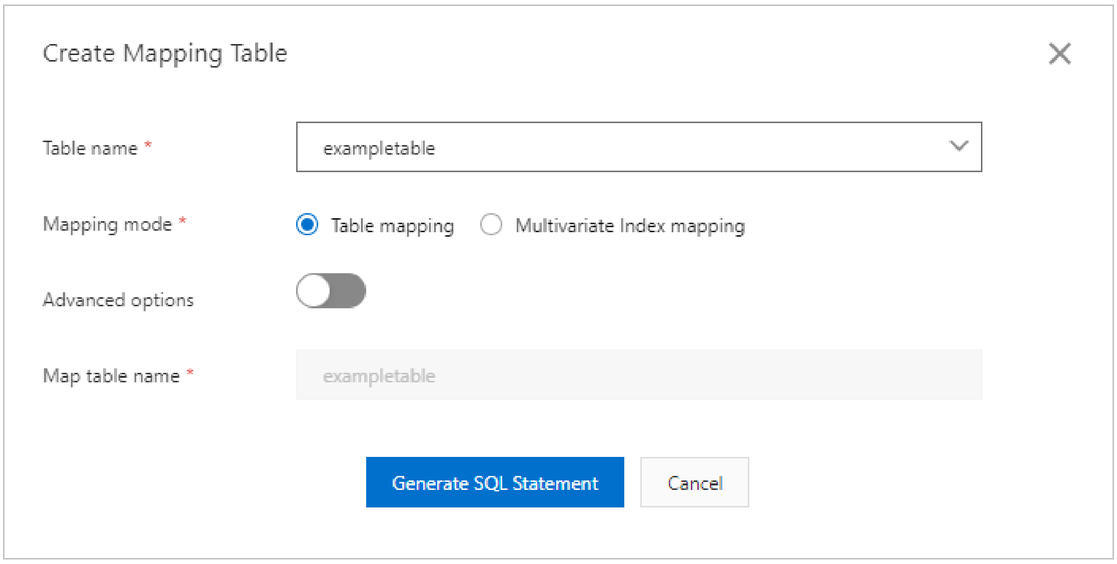
The SQL query feature is available in the China (Hangzhou), China (Shanghai), China (Beijing), China (Zhangjiakou), China (Shenzhen), Germany (Frankfurt), and Singapore regions.

Create a mapping table

1. Log on to the Tablestore console.
2. In the top navigation bar, select a region, for example, China (Hangzhou) or China (Shenzhen).
3. On the **Overview** page, click the name of the required instance or click Manage Instance in the **Actions** column of the required instance.
4. On the **Query by Executing SQL Statement** tab, create a mapping table.

   > **Note** You can also manually compile SQL statements to create a mapping table. For more information, see Create mapping tables for tables.

i. Click the + icon.



Create Mapping Table

Table name * exampletable

Mapping mode * ● Table mapping ○ Multivariate Index mapping

Advanced options

Map table name * exampletable

ii. In the **Create Mapping Table** dialog box, select a table, and click **Generate SQL Statement**.

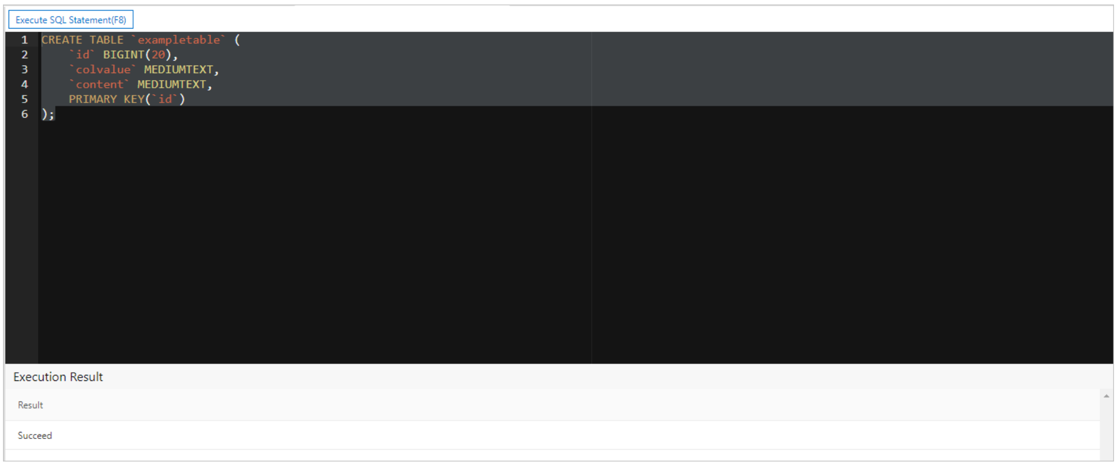
The system automatically generates the schema of the mapping table.

> **Notice** Make sure that the field data types in the mapping table match the field data types in the data table. For more information about data type mappings, see Data type mappings.

iii. After you modify the schema based on your business requirements, hold down the left mouse button to select an SQL statement and click **Execute SQL Statement(F8)**.

After the execution is successful, the execution result is displayed in the **Execution Result** section.

> **Notice**
>
> - Before you execute an SQL statement, you must select the SQL statement that you want to execute. Otherwise, the system executes the first SQL statement by default.
> - You cannot select multiple SQL statements to execute at the same time. If you select multiple SQL statements, the system reports an error.



```
CREATE TABLE `exampletable` (
    `id` BIGINT(20),
    `colvalue` MEDIUMTEXT,
    `content` MEDIUMTEXT,
    PRIMARY KEY(`id`)
);
```

Execution Result

Result

Succeed

Query data

After a mapping table is created, you can execute the SELECT statement to query data on the **Query by Executing SQL Statement** tab. For more information, see Query data.

8.5. Use Tablestore SDKs

You can call the sqlQuery operation to access Tablestore by executing SQL statements.

Prerequisites

-
- An AccessKey pair that consists of an AccessKey ID and an AccessKey secret is obtained. For more information, see Obtain an AccessKey pair.
-
- A Tablestore client is initialized. For more information, see Initialization.

Use Tablestore SDKs

You can use the following Tablestore SDKs to perform a SQL query:

- Java SDK
- Go SDK

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

After you execute the create table statement to create a mapping table for an existing table, you can execute the select statement to query data in the existing table.

1. Execute the create table statement to create a mapping table for an existing table.

   Execute the `create table test_table (pk varchar(1024), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primary key(pk))` statement to create a mapping table for the table named test_table.

   ```
   private static void createTable(SyncClient client) {
       // Create a SQL request.
       SQLQueryRequest request = new SQLQueryRequest("create table test_table (pk varchar(
   1024), long_value bigint, double_value double, string_value mediumtext, bool_value bool
   , primary key(pk))");
       // Obtain the response to the SQL request.
       SQLQueryResponse response = client.sqlQuery(request);
   }
   ```

2. Execute the select statement to query data in the table.

   Execute the `select pk, long_value, double_value, string_value, bool_value from test_table limit 20` statement to query data in the table named test_table and set the maximum number of rows that you want to return to 20. The system returns the request type, the schema of the returned results, and the returned results of the query statement.

```
private static void queryData(SyncClient client) {
    // Create a SQL request.
    SQLQueryRequest request = new SQLQueryRequest("select pk, long_value, double_value,
string_value, bool_value from test_table limit 20");
    // Obtain the response to the SQL request.
    SQLQueryResponse response = client.sqlQuery(request);
    // Obtain the SQL request type.
    System.out.println("response type: " + response.getSQLStatementType());
    // Obtain the schema of the returned results of the SQL request.
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table meta: " + tableMeta.getSchema());
    // Obtain the returned results of the SQL request.
    SQLResultSet resultSet = response.getSQLResultSet();
    System.out.println("response resultset:");
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0) + ", " + row.getString("pk") + ", " +
                           row.getLong(1) + ", " + row.getLong("long_value") + ", " +
                           row.getDouble(2) + ", " + row.getDouble("double_value") + ",
" +
                           row.getString(3) + ", " + row.getString("string_value") + ",
" +
                           row.getBoolean(4) + ", " + row.getBoolean("bool_value"));
    }
}
```

Sample output:

```
response type: SQL_SELECT
response table meta: [pk:STRING, long_value:INTEGER, double_value:DOUBLE, string_value:
STRING, bool_value:BOOLEAN]
response resultset:
binary_null, binary_null, 1, 1, 1.0, 1.0, a, a, false, false
bool_null, bool_null, 1, 1, 1.0, 1.0, a, a, null, null
double_null, double_null, 1, 1, null, null, a, a, true, true
long_null, long_null, null, null, 1.0, 1.0, a, a, true, true
string_null, string_null, 1, 1, 1.0, 1.0, null, null, false, false
```

8.6. Use JDBC

8.6.1. Use JDBC to access Tablestore

This topic describes how to use JDBC to access Tablestore.

Prerequisites

-
- An AccessKey pair that consists of an AccessKey ID and an AccessKey secret is obtained. For more information, see Obtain an AccessKey pair.
- A data table is created, and a mapping table is created for the data table. For more information, see Create a data table and Create mapping tables for tables.

Step 1: Install the JDBC driver

You can install the JDBC driver by using one of the following methods:

- Download the JDBC driver for Tablestore and import the JDBC driver to the project. For more information about the download path, see JDBC driver for Tablestore.
- Add dependencies to a Maven project.

  To use the JDBC driver for Tablestore in Maven, you need only to add the corresponding dependencies to the pom.xml file. In this example, JDBC driver 5.13.5 is used. Add the following content to <dependencies>:

  ```
  <dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>tablestore-jdbc</artifactId>
    <version>5.13.5</version>
  </dependency>
  ```

Step 2: Access Tablestore by using JDBC

1. Load the JDBC driver for Tablestore by using `Class.forName()`.

   The name of the JDBC driver for Tablestore is `com.alicloud.openservices.tablestore.jdbc.OTSDriver`.

   ```
   Class.forName("com.alicloud.openservices.tablestore.jdbc.OTSDriver");
   ```

2. Access a Tablestore instance by using JDBC.

   ```
   String url = "jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance";
   String user = "************************";
   String password = "********************************";
   Connection conn = DriverManager.getConnection(url, user, password);
   ```

   The following table describes the parameters that you must configure to access a Tablestore instance by using JDBC.

Parameter	Example	Description

Parameter	Example	Description
url	jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	The URL of the JDBC driver for Tablestore in the format `jdbc:ots:schema://[accessKeyId:accessKeySecret@]endpoint/instanceName[?param1=value1&...&paramN=valueN]` . The URL contains the following fields: - schema: This field is required and specifies the protocol that is used by the JDBC driver for Tablestore. In most cases, this field is set to https. - accessKeyId:accessKeySecret: This field is optional and specifies the AccessKey ID and AccessKey secret of your Alibaba Cloud account or a RAM user. - endpoint: This field is required and specifies the endpoint of the instance. For more information, see Endpoint. - instanceName: This field is required and specifies the name of the instance. For more information about other configuration items, see Configuration items.
user	************************	The AccessKey ID of your Alibaba Cloud account or a RAM user.
password	********************************	The AccessKey secret of your Alibaba Cloud account or a RAM user.

You can pass the AccessKey pair and configuration items by using a URL or the Properties parameter. The following example shows how to access the myinstance instance in the China (Hangzhou) region over the Internet.

- Pass the AccessKey pair and configuration items by using a URL

  ```
  DriverManager.getConnection("jdbc:ots:https://************************:**************
  ******************@myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance?enableRequestCo
  mpression=true");
  ```

- Pass the AccessKey pair and configuration items by using the Properties parameter

  ```
  Properties info = new Properties();
  info.setProperty("user", "************************");
  info.setProperty("password", "********************************");
  info.setProperty("enableRequestCompression", "true");
  DriverManager.getConnection("jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com
  /myinstance", info);
  ```

3. Execute SQL statements.

   You can use the createStatement or prepareStatement method to create SQL statements.

> **Note** For more information about the supported SQL statements, see SQL features.

- Use the createStatement method to create SQL statements

  ```
  // Create the SQL statement based on your business requirements. The following sample
  code shows how to query the data in the id and name columns in the test_table table:
  String sql = "SELECT id,name FROM test_table";
  Statement stmt = conn.createStatement();
  ResultSet resultSet = stmt.executeQuery(sql);
  while (resultSet.next()) {
      String id = resultSet.getString("id");
      String name = resultSet.getString("name");
      System.out.println(id);
      System.out.println(name);
  }
  resultSet.close();
  stmt.close();
  ```

- Use the prepareStatement method to create SQL statements

  ```
  // Create the SQL statement based on your business requirements. The following sample
  code shows how to query the data with the specified primary key in the test_table tab
  le:
  String sql = "SELECT * FROM test_table WHERE pk = ?";
  PreparedStatement stmt = connection.prepareStatement(sql);
  stmt.setLong(1, 1);
  ResultSet resultSet = stmt.executeQuery();
  ResultSetMetaData metaData = resultSet.getMetaData();
  while (resultSet.next()) {
      int columnCount = metaData.getColumnCount();
      for (int i=0; i< columnCount;i++) {
          String columnName = metaData.getColumnName(i+1);
          String columnValue = resultSet.getString(columnName);
          System.out.println(columnName);
          System.out.println(columnValue);
      }
  }
  resultSet.close();
  stmt.close();
  ```

Complete sample code

The following sample code shows how to query all data in the test_table table of the myinstance instance in the China (Hangzhou) region:

```
public class Demo {
    public static void main(String[] args) throws SQLException, ClassNotFoundException {
        Class.forName("com.alicloud.openservices.tablestore.jdbc.OTSDriver");
        String url = "jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance";
        String user = "************************";
        String password = "********************************";
        Connection conn = DriverManager.getConnection(url, user, password);
        String sql = "SELECT * FROM test_table";
        Statement stmt = conn.createStatement();
        ResultSet resultSet = stmt.executeQuery(sql);
        ResultSetMetaData metaData = resultSet.getMetaData();
        while (resultSet.next()) {
            int columnCount = metaData.getColumnCount();
            for (int i=0; i< columnCount;i++) {
                String columnName = metaData.getColumnName(i+1);
                String columnValue = resultSet.getString(columnName);
                System.out.println(columnName);
                System.out.println(columnValue);
            }
        }
        resultSet.close();
        stmt.close();
        conn.close();    // Close the connection. Otherwise, the program cannot exit.
    }
}
```

Configuration items

The JDBC driver for Tablestore is implemented based on Tablestore SDK for Java. You can use JDBC to modify the configuration items of Tablestore SDK for Java. The following table describes the common configuration items.

Configuration item	Example	Description
enableRequestCompression	false	Specifies whether to compress the request data. Default value: false. Valid values: - true: compresses the request data. - false: does not compress the request data.
enableResponseCompression	false	Specifies whether to compress the response data. Default value: false. Valid values: - true: compresses the response data. - false: does not compress the response data.
ioThreadCount	2	The number of IOReactor threads of the HttpAsyncClient. The default value is the same as the number of vCPUs.

Configuration item	Example	Description
maxConnections	300	The maximum number of allowed HTTP connections.
socketTimeoutInMillisecond	30000	The timeout period for data transmission at the Socket layer. Unit: milliseconds. The value of 0 indicates an indefinite wait.
connectionTimeoutInMillisecond	30000	The timeout period for connection setup. Unit: milliseconds. The value of 0 indicates an indefinite wait.
retryThreadCount	1	The number of threads that are used to execute retries in the thread pool.
syncClientWaitFutureTimeoutInMillis	-1	The timeout period for the asynchronous wait. Unit: milliseconds.
connectionRequestTimeoutInMillisecond	60000	The timeout period for sending the request. Unit: milliseconds.

Data type conversion

Tablestore supports five data types: Integer, Double, String, Binary, and Boolean. When you use Tablestore SDK for Java and JDBC to access Tablestore, the JDBC driver can automatically convert data types between Java and Tablestore.

- Convert Java data types to Tablestore data types

  If you use the PreparedStatement method to specify the values of parameters in SQL statements, the Byte, Short, Int, Long, BigDecimal, Float, Double, String, CharacterStream, Bytes, and Boolean data types in Java can be passed to the SQL engine of Tablestore.

```
PreparedStatement stmt = connection.prepareStatement("SELECT * FROM t WHERE pk = ?");
stmt.setLong(1, 1);                                // The data type can be converted.
stmt.setURL(1, new URL("https://aliyun.com/"));    // The data type cannot be converted,
and the system throws an exception.
```

- Convert Tablestore data types to Java data types

  If you use the ResultSet method to obtain SQL query results, take note of the conversion rules in the following table. The following table describes the rules for the automatic conversion of Tablestore data types to Java data types.

Data type in Tablestore	Conversion rule

Data type in Tablestore	Conversion rule
Integer	◦ When the system converts the data type to an integer type, the system throws an exception if the original value is out of the value range of the integer type. ◦ When the system converts the data type to a floating-point type, the precision of the converted value is lower than the precision of the original value. ◦ When the system converts the data type to the string or binary type, the converted value is the same as the result of processing the original value by using toString(). ◦ When the system converts the data type to the Boolean type and the original value is a non-zero value, the converted value is true.
Double	
String	◦ When the system converts the data type to an integer type or a floating-point type, the system throws an exception if parsing fails. ◦ When the system converts the data type to the Boolean type and the original string is true, the converted value is true.
Binary	
Boolean	◦ When the system converts the data type to an integer type or a floating-point type and the original value is true, the converted value is 1. If the original value is false, the converted value is 0. ◦ When the system converts the data type to the string or binary type, the converted value is the same as the result of processing the original value by using toString().

```
Statement stmt = conn.createStatement();
ResultSet resultSet = stmt.executeQuery("SELECT count(*) FROM t");
while (resultSet.next()) {
    resultSet.getLong(1);                 // The data type can be converted.
    resultSet.getCharacterStream(1);     // The data type cannot be converted, and the sys
tem throws an exception.
}
```

For more information about the data type conversion between Tablestore and Java, see the following table.

Note In the following table, ticks (✓) indicate normal conversion, tildes (~) indicate that an exception may be thrown during conversion, and crosses (×) indicate that the conversion cannot be performed.

Data type	Integer	Double	String	Binary	Boolean
Byte	~	~	~	~	√
Short	~	~	~	~	√
Int	~	~	~	~	√

Data type	Integer	Double	String	Binary	Boolean
Long	√	~	~	~	√
BigDecimal	√	√	~	~	√
Float	√	√	~	~	√
Double	√	√	~	~	√
String	√	√	√	√	√
CharacterStream	×	×	√	√	×
Bytes	√	√	√	√	√
Boolean	√	√	√	√	√

8.6.2. Use Hibernate to use the JDBC driver for Tablestore

This topic describes how to access Tablestore by using Hibernate to use the Java Database Connectivity (JDBC) driver for Tablestore.

Background information

Hibernate is an Object/Relational Mapping (ORM) solution for Java environments. You can use Hibernate to map Java classes to database tables, map Java data types to SQL data types, and query data. Hibernate can significantly reduce the development time that is spent on manually handling data in SQL and JDBC. For more information, see Hibernate documentation.

Prerequisites

-
- An AccessKey pair that consists of an AccessKey ID and an AccessKey secret is obtained. For more information, see Obtain an AccessKey pair.
- A data table is created, and a mapping table is created for the data table. For more information, see Create a data table and Create mapping tables for tables.

Step 1: Install the JDBC driver

You can install the JDBC driver by using one of the following methods:

- Download the JDBC driver for Tablestore and import the JDBC driver to the project. For more information about the download path, see JDBC driver for Tablestore.
- Add dependencies to a Maven project.

  To use the JDBC driver for Tablestore in Maven, you need only to add the corresponding dependencies to the pom.xml file. In this example, JDBC driver 5.13.5 is used. Add the following content to <dependencies>:

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>tablestore-jdbc</artifactId>
  <version>5.13.5</version>
</dependency>
```

Step 2: Install Hibernate

You can install Hibernate by using one of the following methods:

- Download the Hibernate installation package hibernate-core-x.x.x.jar and import the package to the project. For more information about the download path, see Hibernate installation package.

  In hibernate-core-x.x.x.jar, `x.x.x` indicates the version number of Hibernate. Select a Hibernate installation package based on your business requirements.

- Add dependencies to a Maven project.

  To use Hibernate in Maven, you need only to add the corresponding dependencies to the pom.xml file. The following sample code shows how to add content to <dependencies>. In this example, the 3.6.3.Final version is used.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>3.6.3.Final</version>
</dependency>
```

Step 3: Map SQL fields

After you create a Java Bean that contains member variables whose names are the same as the names of fields in the data table, create a mapping configuration file to map the member variables in the Java Bean to the fields in the data table.

1. Create a Java Bean that contains member variables whose names are the same as the names of fields in the data table.

```
package hibernate;
public class Trip {
    private long tripId;
    private long duration;
    private String startDate;
    private String endDate;
    private long startStationNumber;
    private long endStationNumber;
    private String startStation;
    private String endStation;
    private String bikeNumber;
    private String memberType;
    // ...
}
```

2. Create a mapping configuration file to map the member variables in the Java Bean to the fields in the data table. The following sample code shows how to create a mapping configuration file named Trip.hbm.xml in the hibernate directory.

Tablestore SQL does not support data insert and update. Therefore, you must set the insert property and the update property to false. For more information about the mappings between the data types of fields in SQL and Tablestore, see Data type mappings. For more information about the supported SQL features, see SQL features.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!--Specify the actual class name. -->
    <class name="hibernate.Trip" table="trips">
        <!-- The field that is configured in the id element is the primary key column of the data table. -->
        <id name="tripId" column="trip_id" type="long"/>
        <!-- The fields that are configured in the property element are attribute columns of the data table. You must set the insert property and the update property to false because insert and update operations are prohibited for attribute columns. -->
        <property name="duration" column="duration" type="long" insert="false" update="false"/>
        <property name="startDate" column="start_date" type="string" insert="false" update="false"/>
        <property name="endDate" column="end_date" type="string" insert="false" update="false"/>
        <property name="startStationNumber" column="start_station_number" type="long" insert="false" update="false"/>
        <property name="endStationNumber" column="end_station_number" type="long" insert="false" update="false"/>
        <property name="startStation" column="start_station" type="string" insert="false" update="false"/>
        <property name="endStation" column="end_station" type="string" insert="false" update="false"/>
        <property name="bikeNumber" column="bike_number" type="string" insert="false" update="false"/>
        <property name="memberType" column="member_type" type="string" insert="false" update="false"/>
    </class>
</hibernate-mapping>
```

Step 4: Build the SessionFactory

After you configure the Hibernate configuration file, load the Hibernate configuration file to build the SessionFactory.

1. Add the following content to the Hibernate configuration file named hibernate.cfg.xml. Modify the configuration items in the configuration file based on your business requirements.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.alicloud.openservices.tablestore.jdbc.OTSDriver</property>
        <property name="hibernate.connection.url">jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance</property>
        <property name="hibernate.connection.username">************************</property>
        <property name="hibernate.connection.password">********************************</property>
        <property name="hibernate.connection.autocommit">true</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- Specify the path of the mapping configuration file. -->
        <mapping resource="hibernate/Trip.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

The following table describes the items that you must configure in the Hibernate configuration file.

Configuration item	Type	Required	Example	Description
hibernate.connection.driver_class	class	Yes	com.alicloud.openservices.tablestore.jdbc.OTSDriver	The name of the class for the JDBC driver for Tablestore. Set this configuration item to com.alicloud.openservices.tablestore.jdbc.OTSDriver.
hibernate.connection.url	string	Yes	jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	The endpoint of the instance. The value must be in the following format: `jdbc:ots:endpoint/instanceName`. endpoint indicates the endpoint of the instance. For more information, see Endpoint. instanceName indicates the name of the instance. Modify the value of instanceName based on your business requirements. When you specify a value for this configuration item, the `jdbc:ots:` prefix must be included in the value.
hibernate.connection.username	string	Yes	************************	The AccessKey ID of your Alibaba Cloud account or a RAM user.

Configuration item	Type	Required	Example	Description
hibernate.connection.password	string	Yes	********************************	The AccessKey secret of your Alibaba Cloud account or a RAM user.
hibernate.connection.autocommit	boolean	Yes	true	Specifies whether to automatically commit configurations. Tablestore does not support transactions. Set hibernate.connection.autocommit to true.
hibernate.dialect	string	Yes	org.hibernate.dialect.MySQLDialect	Tablestore SQL inherits the MySQL syntax. Set this configuration item to org.hibernate.dialect.MySQLDialect.

2. Load the Hibernate configuration file to build the SessionFactory.

```
SessionFactory factory = new Configuration().
  configure("hibernate/hibernate.cfg.xml").
  buildSessionFactory();
```

Step 5: Create a session to query data

```
Session session = factory.openSession();
Trip trip = (Trip) session.get(Trip.class, 99L);
System.out.println("trip id: " + trip.getTripId());
System.out.println("start date: " + trip.getStartDate());
System.out.println("end date: " + trip.getEndDate());
System.out.println("duration: " + trip.getDuration());
session.close();
factory.close();
```

Complete sample code

The following sample code shows how to query the row in which the value of the primary key is 99 and return the specified columns of the row:

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import hibernate.Trip;
public class HibernateDemo {
    public static void main(String[] args) {
        SessionFactory factory = new Configuration().
                configure("hibernate/hibernate.cfg.xml"). // Specify the full path of the Hibernate configuration file.
                buildSessionFactory();
        Session session = factory.openSession();
        // Set the value of the primary key to 99. If the row in which the value of the primary key is 99 does not exist, null is returned.
        Trip trip = (Trip) session.get(Trip.class, 99L);
        // Display the column values that you want to obtain.
        System.out.println("trip id: " + trip.getTripId());
        System.out.println("start date: " + trip.getStartDate());
        System.out.println("end date: " + trip.getEndDate());
        System.out.println("duration: " + trip.getDuration());
        session.close();
        factory.close();
    }
}
```

FAQ

- Problem description: What do I do if the following error message appears when I query data by using Hibernate to use the JDBC driver?

  ```
  Exception in thread "main" org.hibernate.HibernateException: Unable to instantiate default tuplizer [org.hibernate.tuple.entity.PojoEntityTuplizer]
    at org.hibernate.tuple.entity.EntityTuplizerFactory.constructTuplizer(EntityTuplizerFactory.java:108)
    at org.hibernate.tuple.entity.EntityTuplizerFactory.constructDefaultTuplizer(EntityTuplizerFactory.java:133)
    at org.hibernate.tuple.entity.EntityEntityModeToTuplizerMapping.<init>(EntityEntityModeToTuplizerMapping.java:80)
    at org.hibernate.tuple.entity.EntityMetamodel.<init>(EntityMetamodel.java:322)
    at org.hibernate.persister.entity.AbstractEntityPersister.<init>(AbstractEntityPersister.java:485)
    at org.hibernate.persister.entity.SingleTableEntityPersister.<init>(SingleTableEntityPersister.java:133)
    at org.hibernate.persister.PersisterFactory.createClassPersister(PersisterFactory.java:84)
    at org.hibernate.impl.SessionFactoryImpl.<init>(SessionFactoryImpl.java:286)
    .....
  ```

  Possible cause: The javassist-x.x.x.jar package is missing.

  Solution: Install the javassist-x.x.x.jar package by using one of the following methods:

  - Install the javassist installation package javassist-x.x.x.jar and import the package to the project. For more information about the download path, see javassist installation package.

    In javassist-x.x.x.jar, `x.x.x` indicates the version number of javassist. Select a javassist installation package based on your business requirements.

  - Add dependencies to a Maven project.

    Add the corresponding dependencies to the pom.xml file in the Maven project. The following sample code shows how to add content to <dependencies>. In this example, the 3.15.0-GA version is used.

    ```
    <!-- https://mvnrepository.com/artifact/org.javassist/javassist -->
    <dependency>
        <groupId>org.javassist</groupId>
        <artifactId>javassist</artifactId>
        <version>3.15.0-GA</version>
    </dependency>
    ```

- Problem description: What do I do if the `Message: Unknown column '{columnName}' in 'field list'` error message appears when I query data by using Hibernate to use the JDBC driver?

  Possible cause: The specified column does not exist in the SQL mapping table.

  Solution: Make sure that the specified column exists in the SQL mapping table. You can use one of the following methods to fix the error:

  - Add the specified column to the pre-defined columns to automatically synchronize the specified column to the SQL mapping table.
  - Specify the column when you execute the CREATE TABLE statement to create a mapping table. For more information, see Create mapping tables for tables.

8.6.3. Use MyBatis to use the JDBC driver for Tablestore

This topic describes how to access Tablestore by using MyBatis to use the Java Database Connectivity (JDBC) driver for Tablestore.

Background information

MyBatis is a persistence framework for Java that supports custom SQL statements, stored procedures, and advanced mappings. MyBatis eliminates the need to use JDBC code, manually configure parameters, and retrieve result sets. For more information, see MyBatis documentation.

Prerequisites

-
- An AccessKey pair that consists of an AccessKey ID and an AccessKey secret is obtained. For more information, see Obtain an AccessKey pair.
- A data table is created, and a mapping table is created for the data table. For more information, see Create a data table and Create mapping tables for tables.

Step 1: Install the JDBC driver

You can install the JDBC driver by using one of the following methods:

- Download the JDBC driver for Tablestore and import the JDBC driver to the project. For more information about the download path, see JDBC driver for Tablestore.
- Add dependencies to a Maven project.

  To use the JDBC driver for Tablestore in Maven, you need only to add the corresponding dependencies to the pom.xml file. In this example, JDBC driver 5.13.5 is used. Add the following content to <dependencies>:

  ```
  <dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>tablestore-jdbc</artifactId>
    <version>5.13.5</version>
  </dependency>
  ```

Step 2: Install MyBatis

You can install MyBatis by using one of the following methods:

- Download the MyBatis installation package mybatis-x.x.x.jar and import the package to the project. For more information about the download path, see MyBatis installation package.

  In mybatis-x.x.x.jar, `x.x.x` indicates the version number of MyBatis. Select a MyBatis installation package based on your business requirements.
- Add dependencies to a Maven project.

  To use MyBatis in Maven, you need only to add the corresponding dependencies to the pom.xml file. The following sample code shows how to add content to <dependencies>. In this example, the 3.6.3.Final version is used.

  ```
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.9</version>
  </dependency>
  ```

Step 3: Map SQL fields

1. Create a Java Bean that contains member variables whose names are the same as the names of fields in the data table. In this example, trip_id is the only primary key column in the data table.

   Notice When you create a Java Bean, make sure that the names of the member variables in the Java Bean are the same as the names of the fields in the data table.

```
package mybatis;
public class Trip {
    private long trip_id;
    private long duration;
    private String start_date;
    private String end_date;
    private long start_station_number;
    private long end_station_number;
    private String start_station;
    private String end_station;
    private String bike_number;
    private String member_type;
    // ...
}
```

2. Create a mapping configuration file in which the query conditions are defined. The following sample code shows how to create a mapping configuration file named TripMapper.xml in the mybatis directory.

   For more information about the supported SQL features, see SQL features.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mybatis.TripMapper">
    <select id="selectTrip" resultType="mybatis.Trip">
        select * from trips where trip_id = #{id}
    </select>
</mapper>
```

Step 4: Build the SqlSessionFactory

The SqlSessionFactory is used to create a MyBatis session. You can use a MyBatis session to connect a client to Tablestore.

1. Add the following content to the MyBatis configuration file named mybatis-config.xml. Modify the configuration items in the configuration file based on your business requirements.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <!-- Specify the type of the data source. To exit the process, you must shut down the JDBC driver for Tablestore. Select a type of the data source that matches your needs. -->
            <!-- If you want to keep the program running after a data query, you can set the data source type to POOLED to maintain a connection pool. If you want the program to exit after a data query, you can set the data source type only to UNPOOLED. -->
            <dataSource type="UNPOOLED">
                <property name="driver" value="com.alicloud.openservices.tablestore.jdbc.OTSDriver"/>
                <property name="url" value="jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance"/>
                <property name="username" value="***********************"/>
                <property name="password" value="********************************"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <!-- Specify the path of the mapping configuration file. -->
        <mapper resource="mybatis/TripMapper.xml"/>
    </mappers>
</configuration>
```

The following table describes the items that you must configure in the MyBatis configuration file.

Configuration item	Type	Required	Example	Description
driver	class	Yes	com.alicloud.openservices.tablestore.jdbc.OTSDriver	The name of the class for the JDBC driver for Tablestore. Set this configuration item to com.alicloud.openservices.tablestore.jdbc.OTSDriver.

Configuration item	Type	Required	Example	Description
url	string	Yes	jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	The endpoint of the instance. The value must be in the following format: `jdbc:ots:endpoint/instanceName` . endpoint indicates the endpoint of the instance. For more information, see Endpoint. instanceName indicates the name of the instance. Modify the value of instanceName based on your business requirements. When you specify a value for this configuration item, the `jdbc:ots:` prefix must be included in the value.
username	string	Yes	************************	The AccessKey ID of your Alibaba Cloud account or a RAM user.
password	string	Yes	********************************	The AccessKey secret of your Alibaba Cloud account or a RAM user.

2. Load the MyBatis configuration file to build the SqlSessionFactory.

```
String resource = "mybatis/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream)
;
```

Step 5: Create a SqlSession to query data

```
SqlSession session = sqlSessionFactory.openSession(true);
Trip trip = (Trip) session.selectOne("mybatis.TripMapper.selectTrip", 99L);
System.out.println("trip id: " + trip.getTrip_id());
System.out.println("start date: " + trip.getStart_date());
System.out.println("end date: " + trip.getEnd_date());
System.out.println("duration: " + trip.getDuration());
session.close();
```

Complete sample code

The following sample code shows how to query the row in which the value of the primary key is 99 and return the specified columns of the row:

```
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import mybatis.Trip;
import java.io.IOException;
import java.io.InputStream;
public class MyBatisDemo {
    public static void main(String[] args) throws IOException {
        // Specify the full path of the MyBatis configuration file.
        String resource = "mybatis/mybatis-config.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        // Tablestore does not support transactions. Therefore, you must set the parameter that specifies whether configurations are automatically committed to true.
        SqlSession session = sqlSessionFactory.openSession(true);
        // Enter the identifier of the SELECT statement that you want to execute and set the value of the primary key to 99.
        // The identifier of the SELECT statement must be in the following format: Path of the mapping configuration file.ID of the SELECT statement that you want to execute. In this example, mybatis.TripMapper.selectTrip indicates that the SELECT statement whose ID is selectTrip in the TripMapper.xml file on the mybatis node is executed.
        Trip trip = (Trip) session.selectOne("mybatis.TripMapper.selectTrip", 99L);
        // Display the column values that you want to obtain.
        System.out.println("trip id: " + trip.getTrip_id());
        System.out.println("start date: " + trip.getStart_date());
        System.out.println("end date: " + trip.getEnd_date());
        System.out.println("duration: " + trip.getDuration());
        session.close();
    }
}
```

8.7. Use the Tablestore driver for Go to access Tablestore

This topic describes how to use the Tablestore driver for Go to access Tablestore.

Prerequisites

-
- An AccessKey pair that consists of an AccessKey ID and an AccessKey secret is obtained. For more information, see Obtain an AccessKey pair.
- A data table is created, and a mapping table is created for the data table. For more information, see Create a data table and Create mapping tables for tables.

Step 1: Install the Tablestore driver for Go

Run the following command to install the Tablestore driver for Go:

```
go get github.com/aliyun/aliyun-tablestore-go-sql-driver
```

Step 2: Use the Tablestore driver for Go to access Tablestore

The Tablestore driver for Go is implemented based on the `database/sql/driver` interface. After you import the Tablestore driver for Go package and database/sql, you can use `database/sql` to access Tablestore.

- Parameters

  When you use the Tablestore driver for Go to access Tablestore, you must specify the name of the driver and the name of the Tablestore data source. The following table describes the parameters.

Parameter	Example	Description
driverName	ots	The name of the Tablestore driver for Go. The name is ots and cannot be changed.
dataSourceName	https://*************************.********************************@myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	The name of the Tablestore data source in the following format: `schema://accessKeyId:accessKeySecret@endpoint/instanceName[?param1=value1&...&paramN=valueN]`. The name contains the following fields: ○ schema: required. This field specifies the protocol that is used by the Tablestore driver. In most cases, this field is set to https. ○ accessKeyId:accessKeySecret: required. This field specifies the AccessKey ID and AccessKey secret of your Alibaba Cloud account or a RAM user. ○ endpoint: required. This field specifies the endpoint of the instance. For more information, see Endpoint. ○ instanceName: required. This field specifies the name of the instance. For more information about other configuration items, see Configuration items.

- Sample code

```
import (
    "database/sql"
    _ "github.com/aliyun/aliyun-tablestore-go-sql-driver"
)
// Specify the name of the Tablestore driver for Go and the name of the Tablestore data source.
db, err := sql.Open("ots", "https://access_key_id:access_key_secret@endpoint/instance_name")
if err != nil {
    panic(err)    // Handle errors.
}
```

Step 3: Query data

The Tablestore driver for Go allows you to use the Query method to execute query statements and Prepare to create a statement to query data.

Notice The data types of fields in the query results must match the data types of fields in Tablestore. For more information about data type mappings, see Data type mappings.

- Use the Query method to query data

```
// Create a SQL statement based on your business requirements. The following sample code
provides an example on how to query the data in the pk1, col1, and col2 columns of the te
st_table table:
rows, err := db.Query("SELECT pk1, col1, col2 FROM test_table WHERE pk1 = ?", 3)
if err != nil {
    panic(err)    // Handle errors.
}
for rows.Next() {
    var pk1 int64
    var col1 float64
    var col2 string
    err := rows.Scan(&pk1, &col1, &col2)
    if err != nil {
        panic(err)    // Handle errors.
    }
}
```

- Use Prepare to create a statement to query data

```
// Create a SQL statement based on your business requirements. The following sample code
provides an example on how to query the data in the pk1, col1, and col2 columns of the te
st_table table:
stmt, err := db.Prepare("SELECT pk1, col1, col2 FROM test_table WHERE pk1 = ?")
if err != nil {
    panic(err)    // Handle errors.
}
rows, err := stmt.Query(3)
if err != nil {
    panic(err)    // Handle errors.
}
for rows.Next() {
    var pk1 int64
    var col1 float64
    var col2 string
    err := rows.Scan(&pk1, &col1, &col2)
    if err != nil {
        panic(err)    // Handle errors.
    }
}
```

Complete sample code

The following sample code provides an example on how to query all data in the test_table table on the myinstance instance in the China (Hangzhou) region:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/aliyun/aliyun-tablestore-go-sql"
)
func main() {
    db, err := sql.Open("ots", "https://**********************:************************
******@myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance")
    if err != nil {
        panic(err)
    }
    rows, err := db.Query("SELECT * FROM test_table")
    if err != nil {
        panic(err)
    }
    for rows.Next() {
        // Specify that all columns of the rows that meet the query conditions are returned
.
        columns, err := rows.Columns()
        if err != nil {
            panic(err)
        }
        // Create an array and a pointer to store data.
        values := make([]interface{}, len(columns))
        pointers := make([]interface{}, len(columns))
        for i := range values {
            pointers[i] = &values[i]
        }
        // Scan data rows.
        err = rows.Scan(pointers...)
        if err != nil {
            panic(err)
        }
        fmt.Println(values...)
    }
    rows.Close()
    db.Close()
}
```

Configuration items

You can use the Tablestore driver for Go to modify the configuration items of Tablestore SDK for Go. The following table describes the common configuration items.

Configuration item	Example	Description
retryTimes	10	The allowed number of retries. Default value: 10.
connectionTimeout	15	The timeout period for connection setup. Default value: 15. Unit: seconds. The value of 0 specifies an indefinite period of time.

Configuration item	Example	Description
requestTimeout	30	The timeout period for sending the request. Default value: 30. Unit: seconds.
maxRetryTime	5	The maximum period of time during which retries are triggered. Default value: 5. Unit: seconds.
maxIdleConnections	2000	The maximum number of idle connections. Default value: 2000.

Data type mappings

The following table describes the data type mappings between the fields of Tablestore and the Tablestore driver for Go. If the data types of fields in Tablestore do not match the data types of fields in the Tablestore driver for Go, an error occurs.

Data type in Tablestore	Data type in the Tablestore driver for Go
Integer	int64
Binary	[]byte
String	string
Double	float64
Boolean	bool

8.8. DDL statements

8.8.1. Create mapping tables for tables

You can execute the CREATE TABLE statement to create a mapping table for an existing table or search index. This topic describes how to create a mapping table for an existing table.

Note For more information about how to create a mapping table for an existing search index, see Create mapping tables for search indexes.

Syntax

```
CREATE TABLE [IF NOT EXISTS] table_name(column_name data_type [NOT NULL | NULL],...
| PRIMARY KEY(key_part[,key_part])
)
ENGINE='tablestore',
ENGINE_ATTRIBUTE='{"consistency": consistency [,"allow_inaccurate_aggregation": allow_inacc
urate_aggregation]}';
```

If a table has only one primary key, you can use the following syntax to create a mapping table for the existing table:

```
CREATE TABLE [IF NOT EXISTS] table_name(
column_name data_type PRIMARY KEY,column_name data_type [NOT NULL | NULL],...
)
ENGINE='tablestore',
ENGINE_ATTRIBUTE='{"consistency": consistency [,"allow_inaccurate_aggregation": allow_inacc
urate_aggregation]}';
```

Parameters

Parameter	Required	Description
IF NOT EXISTS	No	Specifies whether a success response is returned. If you specify IF NOT EXISTS, a success response is returned regardless of whether the table exists. Otherwise, a success response is returned only when the table does not exist.
table_name	Yes	The name of the table, which is used to identify the table. The table name in SQL must be the same as the table name in Tablestore.
column_name	Yes	The name of the column. The column name in SQL must be equivalent to the column name in the Tablestore table. For example, if the column name in the Tablestore table is Aa, the column name in SQL must be Aa, AA, aA, or aa.
data_type	Yes	The data type of the column, such as BIGINT, DOUBLE, or BOOL. The data type of the column in SQL must match the data type of the column in the Tablestore table. For more information about data type mappings, see Data type mappings.
NOT NULL | NULL	No	Specifies whether the value of the column can be NULL. Valid values: ● NOT NULL: The value of the column cannot be NULL. By default, the value of a primary key column cannot be NULL. ● NULL: The value of the column can be NULL. By default, the value of an attribute column can be NULL. If the value of an attribute column cannot be NULL, you must set this parameter to NOT NULL for the attribute column.

Parameter	Required	Description
key_part	Yes	The name of the primary key column. Separate multiple primary key columns with commas (,). The name of the primary key column must be included in the column names.
ENGINE	No	The execution engine that is used when you use the mapping table to query data. Default value: tablestore. Valid values: ● tablestore: The SQL engine automatically selects a suitable index to perform the query. ● searchindex: The SQL engine uses the specified search index to perform the query. If ENGINE is set to searchindex, you must configure the index_name and table_name parameters in ENGINE_ATTRIBUTE.

Parameter	Required	Description
ENGINE_ATTRIBUTE	No	The attribute of the execution engine. The value of this parameter is in the JSON format and includes the following items: ● index_name: the name of the search index for which a mapping table is created. You need to specify this item only when a mapping table is created for the search index. ● table_name: the name of the data table for which the search index is created. You need to specify this item only when a mapping table is created for the search index. ● consistency: the consistency mode that is supported by the execution engine. Valid values when you create a mapping table for a table: ○ eventual: The query results are in eventual consistency mode. This is the default value. You can query data a few seconds after the data is written to the table. ○ strong: The query results are in strong consistency mode. You can query data immediately after the data is written to the table. When you create a mapping table for a search index, the value of consistency is eventual and cannot be changed. ● allow_inaccurate_aggregation: specifies whether query performance can be improved by compromising the accuracy of aggregate operations. Type: Boolean. When you create a mapping table for a table, the default value of allow_inaccurate_aggregation is true, which indicates that query performance can be improved by compromising the accuracy of aggregate operations. You can set allow_inaccurate_aggregation to false based on your business requirements. When you create a mapping table for a search index, the value of allow_inaccurate_aggregation is true and cannot be changed.

Examples

- Create a table named exampletable1. The table contains the id primary key column, the colvalue attribute column, and the content attribute column. The id primary key column and colvalue attribute column are of the BIGINT type, and the content attribute column is of the MEDIUMTEXT type.

```
CREATE TABLE exampletable1 (id BIGINT PRIMARY KEY, colvalue BIGINT, content MEDIUMTEXT);
```

- Create a table named exampletable2. The table contains the id primary key column, the colvalue primary key column, and the content attribute column. The id primary key column is of the BIGINT type, the colvalue primary key column is of the VARCHAR type, and the content attribute column is of the MEDIUMTEXT type. The results of queries that are performed on the table must be in strong consistency mode.

```
CREATE TABLE exampletable2 (id BIGINT, colvalue VARCHAR(1024), content MEDIUMTEXT, PRIMAR
Y KEY(colvalue, id)) ENGINE_ATTRIBUTE='{"consistency": "strong"}';
```

8.8.2. Create mapping tables for search indexes

You can execute the CREATE TABLE statement to create a mapping table for an existing table or search index. This topic describes how to create a mapping table for an existing search index.

> **Note** For more information about how to create a mapping table for an existing table, see Create mapping tables for tables.

Background information

Indexes of different types may be created for a data table. When you execute the CREATE TABLE statement to create a mapping table for the data table, and the mapping table is used to query data, the SQL engine automatically selects a data table index, secondary index, or search index to meet your business requirements. You can execute the CREATE TABLE statement in SQL to create a mapping table for a specified search index so that you can select the specified search index to perform the query.

Syntax

```
CREATE TABLE [IF NOT EXISTS] user_defined_name(column_name data_type L,column_name data_typ
e])
ENGINE='searchindex',
ENGINE_ATTRIBUTE='{"index_name": index_name, "table_name": table_name}';
```

Parameters

Parameter	Required	Description
IF NOT EXISTS	No	Specifies whether a success response is returned. If you specify IF NOT EXISTS, a success response is returned regardless of whether the table exists. Otherwise, a success response is returned only when the table does not exist.
user_defined_name	Yes	The name of the mapping table for the search index. The name is used to identify the mapping table in SQL. The name is used for SQL operations.

Parameter	Required	Description
column_name	Yes	The name of the column. The column name in SQL must be equivalent to the column name in the Tablestore table. For example, if the column name in the Tablestore table is Aa, the column name in SQL must be Aa, AA, aA, or aa.
data_type	Yes	The data type of the column, such as BIGINT, DOUBLE, or BOOL. The data type of the column in SQL must match the data type of the column in the Tablestore table. For more information about data type mappings, see Data type mappings.
ENGINE	Yes	The execution engine that is used when you use the mapping table to query data. Default value: tablestore. Valid values: • tablestore: The SQL engine automatically selects a suitable index to perform the query. • searchindex: The SQL engine uses the specified search index to perform the query. If ENGINE is set to searchindex, you must configure the index_name and table_name parameters in ENGINE_ATTRIBUTE.

Parameter	Required	Description
ENGINE_ATTRIBUTE	Yes	The attribute of the execution engine. The value of this parameter is in the JSON format and includes the following items: • index_name: the name of the search index for which a mapping table is created. You need to specify this item only when a mapping table is created for the search index. • table_name: the name of the data table for which the search index is created. You need to specify this item only when a mapping table is created for the search index. • consistency: the consistency mode that is supported by the execution engine. Valid values when you create a mapping table for a table: ○ eventual: The query results are in eventual consistency mode. This is the default value. You can query data a few seconds after the data is written to the table. ○ strong: The query results are in strong consistency mode. You can query data immediately after the data is written to the table. When you create a mapping table for a search index, the value of consistency is eventual and cannot be changed. • allow_inaccurate_aggregation: specifies whether query performance can be improved by compromising the accuracy of aggregate operations. Type: Boolean. When you create a mapping table for a table, the default value of allow_inaccurate_aggregation is true, which indicates that query performance can be improved by compromising the accuracy of aggregate operations. You can set allow_inaccurate_aggregation to false based on your business requirements. When you create a mapping table for a search index, the value of allow_inaccurate_aggregation is true and cannot be changed.

Examples

Create a mapping table named search_exampletable1 for the exampletable1_index search index that is created for the exampletable1 data table. The mapping table contains the id, colvalue, and content columns. The id column is of the BIGINT type, and the colvalue and content columns are of the MEDIUMTEXT type.

```
CREATE TABLE search_exampletable1(id BIGINT, colvalue MEDIUMTEXT, content MEDIUMTEXT) ENGIN
E='searchindex' ENGINE_ATTRIBUTE='{"index_name": "exampletable1_index", "table_name": "exam
pletable1"}';
```

After the search_exampletable1 mapping table is created, you can perform the following operations:

- Query information about the search_exampletable1 mapping table.

  ```
  SHOW INDEX IN search_exampletable1;
  ```

  For more information about how to query index information about a table, see Query the index information about a table.

- Use the search_exampletable1 mapping table to query the rows in which the value of the content column matches at least one of the tokens that are obtained by tokenizing the "tablestore cool" string. Specify that up to 10 rows are returned and the id and content columns are returned in each row that meets the query conditions.

  ```
  SELECT id,content FROM search_exampletable1 WHERE TEXT_MATCH(content, "tablestore cool")
  LIMIT 10;
  ```

  For more information about how to query data that matches the specified string, see Query data and Full-text search.

8.8.3. Delete mapping tables

If the attribute column of a table is modified, you can execute the DROP MAPPING TABLE statement to delete the mapping table of the table and recreate a mapping table for the table. You can delete the mapping tables of multiple tables in a single request.

Note The DROP MAPPING TABLE statement does not delete the tables in Tablestore.

Syntax

```
DROP MAPPING TABLE [IF EXISTS] table_name,...;
```

Parameters

Parameter	Required	Description
table_name	Yes	The name of the table. You can configure multiple table names that are separated with commas (,) in a single request.
IF EXISTS	No	Specifies whether a success response is returned. If you specify IF EXISTS, a success response is returned regardless of whether the mapping table exists. Otherwise, a success response is returned only when the mapping table exists.

Example

Delete the mapping table of a table named exampletable.

```
DROP MAPPING TABLE IF EXISTS exampletable;
```

8.8.4. Query the information about a table

You can execute the DESCRIBE statement to query the information about a table, such as the field name and field type.

Syntax

```
DESCRIBE table_name;
```

Parameters

Parameter	Required	Description
table_name	Yes	The name of the table.

Examples

Query the information about a table named exampletable.

```
DESCRIBE exampletable;
```

8.9. DQL statements

8.9.1. Query data

You can execute the SELECT statement to query data in a table.

Usage notes

The execution priority of clauses in the SELECT statement is WHERE > GROUP BY > HAVING > ORDER BY > LIMIT and OFFSET.

Syntax

```
SELECT
    [ALL | DISTINCT | DISTINCTROW]
    select_expr [, select_expr] ...
    [FROM table_references]
    [WHERE where_condition]
    [GROUP BY groupby_condition]
    [HAVING having_condition]
    [ORDER BY order_condition]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

Parameters

Parameter	Required	Description
ALL | DISTINCT | DISTINCTROW	No	Specifies whether to remove duplicate fields. Default value: ALL. Valid values: - ALL: returns all values of the specified fields, including duplicate values. - DISTINCT: removes duplicate fields and returns only the values of distinct fields. - DISTINCTROW: removes duplicate rows and returns only the values of distinct rows.
select_expr	Yes	The name or expression of the column in the `column_name[, column_name][, column_exp],..` format. For more information, see Column expression (select_expr).
table_references	Yes	The information about the table whose data you want to query. The value of this parameter can be a table name or a SELECT statement in the `table_name | select_statement` format. For more information, see Table information (table_references).
where_condition	No	The WHERE clause that can be used together with different conditions to implement specific features. - You can use the WHERE clause together with relational operators to query the data that meets specified conditions. The format is `column_name operator value [AND | OR] [column_name operator value]`. For more information, see WHERE clause (where_condition). - You can use the WHERE clause together with the conditions of match query or match phrase query to perform full-text search. For more information, see Full-text search.
groupby_condition	No	The GROUP BY clause that can be used together with aggregate functions. The format is `column_name`. For more information, see GROUP BY clause (groupby_condition).
having_condition	No	The HAVING clause that can be used together with aggregate functions. The format is `aggregate_function(column_name) operator value`. For more information, see HAVING clause (having_condition).
order_condition	No	The ORDER BY clause in the `column_name [ASC | DESC][,column_name [ASC | DESC],...]` format. For more information, see ORDER BY clause (order_condition).

Parameter	Required	Description
row_count	No	The maximum number of rows to return in the query.
offset	No	The data that is skipped in the query. Default value: 0.

Column expression (select_expr)

You can use select_expr to specify the column that you want to query. When you use select_expr, take note of the following items:

- You can use the wildcard character (*) to query all columns. You can also use the WHERE clause to specify the query condition.

  ```
  SELECT * FROM orders;
  ```

  The following example shows how to use the WHERE clause to specify a query condition:

  ```
  SELECT * FROM orders WHERE orderprice >= 100;
  ```

- You can use the column name to specify the column that you want to query.

  ```
  SELECT username FROM orders;
  ```

Table information (table_references)

You can use table_references to specify the table whose data you want to query.

```
SELECT orderprice FROM orders;
```

WHERE clause (where_condition)

You can use where_condition to query data that meets the specified conditions. When you use where_condition, take note of the following items:

- You can use simple expressions that are constructed by using operators such as arithmetic operators or relational operators.

  ```
  SELECT * FROM orders WHERE username = 'lily';
  SELECT * FROM orders WHERE orderprice >= 100;
  ```

- You can use combined expressions that are constructed by using logical operators.

  ```
  SELECT * FROM orders WHERE username = 'lily' AND orderprice >= 100;
  ```

For more information about operators, see SQL operators.

GROUP BY clause (groupby_condition)

You can use groupby_condition to group the row data in the result set of a SELECT statement based on a specified condition. When you use groupby_condition, take note of the following items:

- You can group row data by field.

```
SELECT username FROM orders GROUP BY username;
```

- You can use aggregate functions on grouped columns.

```
SELECT username,COUNT(*) FROM orders GROUP BY username;
```

- You must add the columns that do not use aggregate functions in the SELECT statement to the GROUP BY clause.

```
SELECT username,orderprice FROM orders GROUP BY username,orderprice;
```

For more information about aggregate functions, see Aggregate functions.

HAVING clause (having_condition)

You can use having_condition to filter the row data that you grouped in the result sets that are obtained by using the WHERE and GROUP BY clauses. The row data that you grouped is filtered based on specified conditions.

In most cases, the HAVING clause is used together with aggregate functions to filter data.

```
SELECT username,SUM(orderprice) FROM orders GROUP BY username HAVING SUM(orderprice) < 500;
```

ORDER BY clause (order_condition)

You can use order_condition to sort the row data in the result set of a query based on the specified field and sorting method. When you use order_condition, take note of the following items:

- You can use the ASC or DESC keyword to specify the sorting method. By default, the data is sorted in ascending order (ASC).

```
SELECT * FROM orders ORDER BY orderprice DESC LIMIT 10;
```

- You can specify multiple fields based on which you want to sort data.

```
SELECT * FROM orders ORDER BY username ASC,orderprice DESC LIMIT 10;
```

- You can use LIMIT to limit the number of rows to return.

```
SELECT * FROM orders ORDER BY orderprice LIMIT 10;
```

8.9.2. Aggregate functions

This topic describes the aggregate functions that are supported for SQL queries.

Function	Description
COUNT()	Returns the number of rows that match the specified condition.
COUNT(DISTINCT)	Returns the number of rows with different values in the specified column.
SUM()	Returns the sum of numeric columns.
AVG()	Returns the average value of numeric columns.

Function	Description
MAX()	Returns the maximum value in a column.
MIN()	Returns the minimum value in a column.

8.9.3. Full-text search

To perform full-text search, you can use the conditions of match query (TEXT_MATCH) or match phrase query (TEXT_MATCH_PHRASE) as the WHERE clause in the SELECT statement to query data that matches the specified character string in the table.

Prerequisites

A search index is created for the table whose data you want to query, and tokenization is performed for the column that you want to query. For more information, see Create search indexes.

Note For more information about tokenization, see Tokenization.

Match query

This query uses approximate matches to retrieve query results. Tablestore tokenizes the values in TEXT columns and the keywords that you use to perform match queries based on the analyzer that you specify. This way, Tablestore can perform match queries based on the tokens. We recommend that you use TEXT_MATCH_PHRASE to achieve high performance when you perform fuzzy query to query columns for which fuzzy tokenization is performed.

- SQL expression

```
TEXT_MATCH(fieldName, text, [options])
```

- Parameters

Parameter	Type	Required	Example	Description
fieldName	string	Yes	col1	The name of the column that you want to query. You can perform match queries on TEXT columns.

Parameter	Type	Required	Example	Description
text	string	Yes	"tablestore is cool"	The keyword that is used to match the column values when you perform a match query. If the column to query is a TEXT column, the keyword is tokenized into multiple tokens based on the analyzer that you specify when you create the search index. By default, single-word tokenization is performed if you do not specify the analyzer when you create the search index. For example, if you set the tokenization method to single-word tokenization and use "this is" as a search keyword, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
options	string	No	"or", "2"	The options that you want to use to perform match queries. Valid values: ○ operator: the logical operator. Valid values: OR and AND. Default value: OR. ○ minimum_should_match: the minimum number of matched tokens that are contained in a column value. Default value: 1. If you set operator to OR, a row meets the query conditions only if the value of the fieldName column in the row contains at least the minimum number of matched tokens. If you set operator to AND, a row meets the query conditions only if the column value contains all tokens.

- Return value

  The return value indicates whether the row meets the query conditions. The return value is of the Boolean type. If the return value is true, the row meets the query conditions. If the return value is false, the row does not meet the query conditions.

- Examples

  The following sample code shows how to query the rows in which the value of the col1 column matches at least two tokens of the "tablestore is cool" string in the exampletable table:

```
SELECT * FROM exampletable WHERE TEXT_MATCH(col1, "tablestore is cool", "or", "2")
```

  The following sample code shows how to query the rows in which the value of the col1 column matches all tokens of the "tablestore is cool" string in the exampletable table:

```
SELECT * FROM exampletable WHERE TEXT_MATCH(col1, "tablestore is cool", "and")
```

Match phrase query

Match phrase query is similar to match query (TEXT_MATCH), but evaluates the position between multiple tokens. A row meets the query conditions only when the order and positions of the tokens in the row match the order and positions of the tokens that are contained in the tokenized keyword.

- SQL expression

```
TEXT_MATCH_PHRASE(fieldName, text)
```

- Parameters

Parameter	Type	Required	Example	Description
fieldName	string	Yes	col1	The name of the column that you want to query. You can perform match phrase queries on TEXT columns.
text	string	Yes	"tablestore is cool"	The keyword that is used to match the column values when you perform a match phrase query. If the column to query is a TEXT column, the keyword is tokenized into multiple tokens based on the analyzer that you specify when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you query the phrase "this is", "..., this is tablestore" and "this is a table" are returned. "this table is ..." and "is this a table" are not returned.

- Return value

The return value indicates whether the row meets the query conditions. The return value is of the Boolean type. If the return value is true, the row meets the query conditions. If the return value is false, the row does not meet the query conditions.

- Example

  The following sample code shows how to query the rows in which the value of the col1 column matches the "tablestore is cool" string in the exampletable table:

  ```
  SELECT * FROM exampletable WHERE TEXT_MATCH_PHRASE("col1", "tablestore is cool")
  ```

8.10. Database administration statements

8.10.1. Query the index information about a table

You can execute the SHOW INDEX statement to query the index information about a table.

Syntax

```
SHOW INDEX {FROM | IN} table_name;
```

Parameters

Parameter	Required	Description
table_name	Yes	The name of the table.

Example

Query the index information about a table named exampletable.

```
SHOW INDEX IN exampletable;
```

8.10.2. List table names

You can execute the SHOW TABLES statement to list the names of tables in the current database.

Syntax

```
SHOW TABLES;
```

Examples

List the names of tables in the current database.

```
SHOW TABLES;
```

The following result is returned:

```
Tables_in_tpch
exampletable1
exampletable2
```

8.11. Query optimization

8.11.1. Index selection policy

Tablestore can store large amounts of structured data and supports various types of index schemas for accelerated query and analytics in different scenarios. When you use the SQL query feature, you can perform index-based queries by using explicit access to a secondary index table. Tablestore provides the following methods to query data by using a search index: automatic selection of a search index and explicit access to a search index.

> **Note** For more information about secondary indexes and search indexes, see Overview of secondary indexes and Overview of search indexes.

Use secondary index tables

> **Notice** Secondary indexes cannot be automatically selected for data queries.

Tablestore supports only explicit access to secondary index tables. If you want to access a secondary index table by using explicit access to the secondary index table, perform the following steps:

1. Execute the CREATE TABLE statement to create a mapping table for the secondary index table that you want to access. For more information, see Create mapping tables for tables.
2. Execute the SELECT statement to query data. For more information, see Query data.

Use search indexes

When you use SQL to perform complex queries such as queries based on non-primary key columns and Boolean queries, we recommend that you create a search index for the data table whose data you want to query. After the search index is created, you can use one of the following methods to query data by using the search index in SQL:

- Explicit access to a search index

  If you want to use the specified search index to query data, you can perform the following operations to use this method:

  i. Execute the CREATE TABLE statement to create a mapping table for the search index. For more information, see Create mapping tables for search indexes.

  ii. Execute the SELECT statement to query data. For more information, see Query data.

- Automatic selection of a search index

> **Notice** If you specify that the query results must be in strong consistency mode or query performance cannot be improved by compromising the accuracy of aggregate operations when you create a mapping table for the table whose data you want to query, Tablestore does not automatically select a search index for data query.

When the search index that you want to access is not explicitly specified, if all filtering columns in the WHERE clause and return columns in the SELECT statement are contained in a search index, Tablestore automatically selects the search index for data query. For example, in the `Select A,B,C from exampletable where A=XXX and D = YY;` statement, if the A, B, C, and D columns are contained in a search index of the exampletable table, Tablestore automatically selects the search index for data query.

If SQL statements that combine GROUP BY clauses and aggregate functions match the aggregation capability of the Search API operation of a search index, Tablestore also identifies operators and pushes them down to the search index. For more information about operator pushdown, see Computing pushdown.

Mapping between SQL expressions and Search queries in search indexes

SQL expression	Example	Search query
without predicate	N/A	Match all query
=	• a = 1 • b = "hello world"	Term query
>	a > 1	Range query
>=	a >= 2	
<	a < 5	
<=	a <= 10	
is null	a is null	Exists query
is not null	a is not null	
and	a = 1 and b = "hello world"	Boolean query
or	a > 1 or b = 2	
not	not a = 1	
!=	a !=1	
like	a like "%s%"	Wildcard query
in	a in (1,2,3)	Terms query
text_match	text_match("a", "tablestore cool")	Match query

SQL expression	Example	Search query
text_match_phrase	text_match_phrase("a", "tablestore cool")	Match phrase query

8.11.2. Computing pushdown

A search index provides features such as conditional filtering, aggregation, and sorting. After you create a search index, the system can make full use of the computing power of the search index and push some SQL computing tasks down to the search index for execution. This avoids full table scans and improves computing efficiency.

Prerequisites

A search index is created. For more information about how to create a search index, see Create search indexes.

Scenarios

If a search index contains the data columns involved in SQL statements, the SQL engine reads data by using the search index and pushes down the operators that are supported by the search index. For example, a table named exampletable has a, b, c, and d columns. A search index of the table contains the b, c, and d columns that are indexed. The SQL engine reads data by using the search index when only the b, c, and d columns are involved in SQL statements.

```
SELECT a, b, c, d FROM exampletable; /* The search index does not contain the a column. The
SQL engine reads data by scanning the entire table and does not push down operators. */
SELECT b, c, d FROM exampletable;    /* The search index contains the b, c, and d columns.
The SQL engine reads data by using the search index and pushes down operators. */
```

Operators that can be pushed down

Type	Operator	Pushdown limit
Logical operators	AND and OR	The NOT operator cannot be pushed down.

Type	Operator	Pushdown limit

Relational operators	=, !=, <, <=, >, >=, and BETWEEN ... AND ...	Operator pushdown is supported only for comparison between data columns and constants. Operator pushdown is not supported for comparison between data columns.
		`SELECT * FROM exampletable WHERE a > 1;`
		`/* Operator pushdown is supported for comparison between data columns and constants. */`
		`SELECT * FROM exampletable WHERE a > b;`
		`/* Operator pushdown is not supported for comparison between data columns. */`

Aggregate functions	- Basic aggregation: MIN, MAX, COUNT, AVG, SUM, and ANY_VALUE
	- Deduplication and aggregation: COUNT(DISTINCT col_name)
	- Grouping: GROUP BY col_name
		An aggregate function can aggregate all data or data in a GROUP BY group. Operator pushdown is supported only when the aggregate function supports pushdown and the function parameter is a data column.
		`SELECT COUNT(*) FROM exampletable;`
		`/* Operator pushdown is supported for the special usage of COUNT(*). */`
		`SELECT SUM(a) FROM exampletable;`
		`/* Operator pushdown is supported when the function parameter is a data column. */`
		`SELECT a, b FROM exampletable GROUP BY a, b; /* Operator pushdown is supported for grouping by data column. */`
		`SELECT a FROM exampletable GROUP BY a+1;`
		`/* Operator pushdown is not supported for grouping by expression. */`
		`SELECT SUM(a+b) FROM exampletable;`
		`/* Operator pushdown is not supported when the function parameter is an expression. */`

Type	Operator	Pushdown limit

LIMIT	• LIMIT row_count
	• ORDER BY col_name LIMIT row_count
		Operator pushdown is supported only when the parameter in the ORDER BY clause is a data column.
		`SELECT * FROM exampletable ORDER BY a LIMIT 1;     /* Operator pushdown is supported for sorting by data column. */`
		`SELECT * FROM exampletable ORDER BY a, b LIMIT 1;   /* Operator pushdown is supported for sorting by data column. */`
		`SELECT * FROM exampletable ORDER BY a+1 LIMIT 1;    /* Operator pushdown is not supported for sorting by expression. */`

8.12. Appendix

8.12.1. SQL operators

This topic describes the operators that are supported in Tablestore SQL, including arithmetic operators, relational operators, logical operators, and bitwise operators.

Arithmetic operators

Arithmetic operators can be used in SELECT or WHERE clauses to compute values.

Operator	Relation	Description
A+B	Addition	Returns the result by adding A and B.
A-B	Subtraction	Returns the result by subtracting B from A.
A*B	Multiplication	Returns the result by multiplying A by B.
A/B or A DIV B	Division	Returns the result by dividing A by B.
A%B or A MOD B	Remainder	Returns the result by computing the remainder of dividing A by B.

Relational operators

Relational operators are used to determine the row data that meets the specified conditions in a table.

- If the comparison result is true (TRUE), 1 is returned.
- If the comparison result is false (FALSE), 0 is returned.

Relational operators can be used in WHERE clauses as specific conditions. If the condition is met, 1 is returned. If the condition is not met, 0 is returned.

Operator	Relation	Description
A:=B	Assignment	Assigns the value of B to A.
A=B	Equal to	Returns 1 if A is equal to B, and returns 0 in other cases.
A!=B or A<>B	Not equal to	Returns 1 if A is not equal to B, and returns 0 in other cases.
A>B	Greater than	Returns 1 if A is greater than B, and returns 0 in other cases.
A<B	Less than	Returns 1 if A is less than B, and returns 0 in other cases.
A>=B	Greater than or equal to	Returns 1 if A is greater than or equal to B, and returns 0 in other cases.
A<=B	Less than or equal to	Returns 1 if A is less than or equal to B, and returns 0 in other cases.
BETWEEN A AND B	BETWEEN	Returns 1 if the value is greater than or equal to A and less than or equal to B, and returns 0 in other cases.
Not BETWEEN A AND B	NOT BETWEEN	Returns 1 if the value is greater than B or less than A, and returns 0 in other cases.
A LIKE B	LIKE	Returns 1 if A matches B, and returns 0 in other cases. The LIKE operator performs the string matching operation. A is a string, and B is a matching pattern. The underscore (_) wildcard in the pattern substitutes for exactly one character in a string. The percent sign (%) wildcard in the pattern substitutes for zero or more characters in a string.
A NOT LIKE B	NOT LIKE	Returns 1 if A does not match B, and returns 0 in other cases. The NOT LIKE operator performs the string mismatching operation. A is a string, and B is a matching pattern. The underscore (_) wildcard in the pattern substitutes for exactly one character in a string. The percent sign (%) wildcard in the pattern substitutes for one or more characters in a string.

Logical operators

Logical operators are used to determine whether expressions are true or false.

- If the expression is true (TRUE), 1 is returned.
- If the expression is false (FALSE), 0 is returned.

Logical operators can be used in WHERE clauses to construct complex conditions. If the condition is met, 1 is returned. If the condition is not met, 0 is returned.

Operator	Relation	Description
A AND B or A&&B	Logical AND	Returns 1 if both A and B are TRUE, and returns 0 in other cases.
A OR B	Logical OR	Returns 1 if at least one of A and B is TRUE, and returns 0 in other cases.
A XOR B	Logical XOR	Returns 1 if A and B are not TRUE or FALSE at the same time, and returns 0 in other cases.
NOT A or ! A	Logical NOT	Returns 1 if A is FALSE, and returns 0 in other cases.

Bitwise operators

Bitwise operators are used to compute binary data. The bitwise operation converts the operand into a binary number for bitwise operation, and then converts the computing result from a binary number to a decimal number.

Operator	Relation	Description
A&B	Bitwise AND	Returns the result based on the bitwise AND operation of A and B.
A|B	Bitwise OR	Returns the result based on the bitwise OR operation of A and B.
A^B	Bitwise XOR	Returns the result based on the bitwise XOR operation of A and B.
~A	Bitwise NOT	Returns the result based on the bitwise inversion of A.

8.12.2. Reserved words and keywords

This topic describes all reserved words and keywords in Tablestore SQL.

Alphabetical order	Keyword and reserved word
A	ACCESSIBLE ACCOUNT ACTION ADD AFTER AGAINST AGGREGATE ALGORITHM ALL ALTER ALWAYS ANALYSE ANALYZE ANDANY AS ASC ASCII ASENSITIVE AT AUTOEXTEND_SIZE AUTO_INCREMENT AVG AVG_ROW_LENGTH
B	BACKUP BEFORE BEGIN BETWEEN BIGINT BINARY BINLOG BIT BLOB BLOCK BOOL BOOLEAN BOTH BTREE BY BYTE

Alphabetical order	Keyword and reserved word
C	CACHE CALL CASCADE CASCADED CASE CATALOG_NAME CHAIN CHANGE CHANGED CHANNEL CHAR CHARACTER CHARSET CHECK CHECKSUM CIPHER CLASS_ORIGIN CLIENT CLOSE COALESCE CODE COLLATE COLLATION COLUMN COLUMNS COLUMN_FORMAT COLUMN_NAME COMMENT COMMIT COMMITTED COMPACT COMPLETION COMPRESSED COMPRESSION CONCURRENT CONDITION CONNECTION CONSISTENT CONSTRAINT CONSTRAINT_CATALOG CONSTRAINT_NAME CONSTRAINT_SCHEMA CONTAINS CONTEXT CONTINUE CONVERT CPU CREATE CROSS CUBE CURRENT CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP CURRENT_USER CURSOR CURSOR_NAME
D	DATA DATABASE DATABASES DATAFILE DATE DATETIME DAY DAY_HOUR DAY_MICROSECOND DAY_MINUTE DAY_SECOND DEALLOCATE DEC DECIMAL DECLARE DEFAULT DEFAULT_AUTH DEFINER DELAYED DELAY_KEY_WRITE DELETE DESC DESCRIBE DES_KEY_FILE DETERMINISTIC DIAGNOSTICS DIRECTORY DISABLE DISCARD DISK DISTINCT DISTINCTROW DIV DO DOUBLE DROP DUAL DUMPFILE DUPLICATE DYNAMIC
E	EACH ELSE ELSEIF ENABLE ENCLOSED ENCRYPTION END ENDS ENGINE ENGINES ENUM ERROR ERRORS ESCAPE ESCAPED EVENT EVENTS EVERY EXCHANGE EXECUTE EXISTS EXIT EXPANSION EXPIRE EXPLAIN EXPORT EXTENDED EXTENT_SIZE
F	FALSE FAST FAULTS FETCH FIELDS FILE FILE_BLOCK_SIZE FILTER FIRST FIXED FLOAT FLOAT4 FLOAT8 FLUSH FOLLOWS FOR FORCE FOREIGN FORMAT FOUND FROM FULL FULLTEXT FUNCTION
G	GENERAL GENERATED GEOMETRY GEOMETRYCOLLECTION GET GET_FORMAT GLOBAL GRANT GRANTS GROUP GROUP_REPLICATION
H	HANDLER HASH HAVING HELP HIGH_PRIORITY HOST HOSTS HOUR HOUR_MICROSECOND HOUR_MINUTE HOUR_SECOND
I	IDENTIFIED IF IGNORE IGNORE_SERVER_IDS IMPORT IN INDEX INDEXES INFILE INITIAL_SIZE INNER INOUT INSENSITIVE INSERT INSERT_METHOD INSTALL INSTANCE INT INT1 INT2 INT3 INT4 INT8 INTEGER INTERVAL INTO INVOKER IO IO_AFTER_GTIDS IO_BEFORE_GTIDS IO_THREAD IPC IS ISOLATION ISSUER ITERATE
J	JOIN JSON
K	KEY KEYS KEY_BLOCK_SIZE KILL
L	LANGUAGE LAST LEADING LEAVE LEAVES LEFT LESS LEVEL LIKE LIMIT LINEAR LINES LINESTRING LIST LOAD LOCAL LOCALTIME LOCALTIMESTAMP LOCK LOCKS LOGFILE LOGS LONG LONGBLOB LONGTEXT LOOP LOW_PRIORITY
M	MASTER MASTER_AUTO_POSITION MASTER_BIND MASTER_CONNECT_RETRY MASTER_DELAY MASTER_HEARTBEAT_PERIOD MASTER_HOST MASTER_LOG_FILE MASTER_LOG_POS MASTER_PASSWORD MASTER_PORT MASTER_RETRY_COUNT MASTER_SERVER_ID MASTER_SSL MASTER_SSL_CA MASTER_SSL_CAPATH MASTER_SSL_CERT MASTER_SSL_CIPHER MASTER_SSL_CRL MASTER_SSL_CRLPATH MASTER_SSL_KEY MASTER_SSL_VERIFY_SERVER_CERT MASTER_TLS_VERSION MASTER_USER MATCH MAXVALUE MAX_CONNECTIONS_PER_HOUR MAX_QUERIES_PER_HOUR MAX_ROWS MAX_SIZE MAX_UPDATES_PER_HOURMAX_USER_CONNECTIONS MEDIUM MEDIUMBLOB MEDIUMINT MEDIUMTEXT MEMORY MERGE MESSAGE_TEXT MICROSECOND MIDDLEINT MIGRATE MINUTE MINUTE_MICROSECOND MINUTE_SECOND MIN_ROWS MOD MODE MODIFIES MODIFY MONTH MULTILINESTRING MULTIPOINT MULTIPOLYGON MUTEX MYSQL_ERRNO

Alphabetical order	Keyword and reserved word
N	NAME NAMES NATIONAL NATURAL NCHAR NDB NDBCLUSTER NEVER NEW NEXT NO NODEGROUP NONBLOCKING NONE NOT NO_WAIT NO_WRITE_TO_BINLOG NULL NUMBER NUMERIC NVARCHAR
O	OFFSET OLD_PASSWORD ON ONE ONLY OPEN OPTIMIZE OPTIMIZER_COSTS OPTION OPTIONALLY OPTIONS OR ORDER OUT OUTER OUTFILE OWNER
P	PACK_KEYS PAGE PARSER PARSE_GCOL_EXPR PARTIAL PARTITION PARTITIONING PARTITIONS PASSWORD PHASE PLUGIN PLUGINS PLUGIN_DIR POINT POLYGON PORT PRECEDES PRECISION PREPARE PRESERVE PREV PRIMARY PRIVILEGES PROCEDURE PROCESSLIST PROFILE PROFILES PROXY PURGE
Q	QUARTER QUERY QUICK
R	RANGE READ READS READ_ONLY READ_WRITE REAL REBUILD RECOVER REDOFILE REDO_BUFFER_SIZE REDUNDANT REFERENCES REGEXP RELAY RELAYLOG RELAY_LOG_FILE RELAY_LOG_POS RELAY_THREAD RELEASE RELOAD REMOVE RENAME REORGANIZE REPAIR REPEAT REPEATABLE REPLACE REPLICATE_DO_DB REPLICATE_DO_TABLE REPLICATE_IGNORE_DB REPLICATE_IGNORE_TABLE REPLICATE_REWRITE_DB REPLICATE_WILD_DO_TABLE REPLICATE_WILD_IGNORE_TABLE REPLICATION REQUIRE RESET RESIGNAL RESTORE RESTRICT RESUME RETURN RETURNED_SQLSTATE RETURNS REVERSE REVOKE RIGHT RLIKE ROLLBACK ROLLUP ROTATE ROUTINE ROW ROWS ROW_COUNT ROW_FORMAT RTREE
S	SAVEPOINT SCHEDULE SCHEMA SCHEMAS SCHEMA_NAME SECOND SECOND_MICROSECOND SECURITY SELECT SENSITIVE SEPARATOR SERIAL SERIALIZABLE SERVER SESSION SET SHARE SHOW SHUTDOWN SIGNAL SIGNED SIMPLE SLAVE SLOW SMALLINT SNAPSHOT SOCKET SOME SONAME SOUNDSSOURCE SPATIAL SPECIFIC SQL SQLEXCEPTION SQLSTATE SQLWARNING SQL_AFTER_GTIDS SQL_AFTER_MTS_GAPS SQL_BEFORE_GTIDS SQL_BIG_RESULT SQL_BUFFER_RESULT SQL_CACHE SQL_CALC_FOUND_ROWS SQL_NO_CACHE SQL_SMALL_RESULT SQL_THREAD SQL_TSI_DAY SQL_TSI_HOUR SQL_TSI_MINUTE SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_SECOND SQL_TSI_WEEK SQL_TSI_YEAR SSL STACKED START STARTING STARTS STATS_AUTO_RECALC STATS_PERSISTENT STATS_SAMPLE_PAGES STATUS STOP STORAGE STORED STRAIGHT_JOIN STRING SUBCLASS_ORIGIN SUBJECT SUBPARTITION SUBPARTITIONS SUPER SUSPEND SWAPS SWITCHES
T	TABLE TABLES TABLESPACE TABLE_CHECKSUM TABLE_NAME TEMPORARY TEMPTABLE TERMINATED TEXT THAN THEN TIME TIMESTAMP TIMESTAMPADD TIMESTAMPDIFF TINYBLOB TINYINT TINYTEXT TO TRAILING TRANSACTION TRIGGER TRIGGERS TRUE TRUNCATE TYPE TYPES
U	UNCOMMITTED UNDEFINED UNDO UNDOFILE UNDO_BUFFER_SIZE UNICODE UNINSTALL UNION UNIQUE UNKNOWN UNLOCK UNSIGNED UNTIL UPDATE UPGRADE USAGE USE USER USER_RESOURCES USE_FRM USING UTC_DATE UTC_TIME UTC_TIMESTAMP
V	VALIDATION VALUE VALUES VARBINARY VARCHAR VARCHARACTER VARIABLES VARYING VIEW VIRTUAL
W	WAIT WARNINGS WEEK WEIGHT_STRING WHEN WHERE WHILE WITH WITHOUT WORK WRAPPER WRITE
X	X509 XA XID XML XOR
Y	YEAR YEAR_MONTH

Alphabetical order	Keyword and reserved word
Z	ZEROFILL

9.Tunnel service

9.1. Overview

Tunnel Service is a centralized service that uses the Table Store API to allow you to consume full and incremental data. Tunnel Service provides tunnels that are used to export and consume data in the full, incremental, and differential modes. After you create a tunnel, you can use it to consume historical and incremental data that is exported from a specified table.

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

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

9.2. Features

Tunnel Service provides tunnels for full and incremental data consumption, orderly incremental data consumption, consumption latency monitoring, and horizontal scaling of data consumption capabilities.

> **Note** In scenarios where a table is written for 100,000 times per seconds, Tunnel Service provides a latency of milliseconds from the data is updated until the update record is obtained. The update record is returned in the sequence that the data is updated.

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 sequentially distributes incremental data to one or more logical partitions 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 latency (time point) of the consumed data on each client. Tunnel Service also allows you to monitor data consumption of tunnels in the Tablestore 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.

9.3. Tunnel clients

A tunnel client is an automatic data consumption framework of Tunnel Service. Before using Tunnel Service, you must know the following features of tunnel clients: automatic data processing, automatic load balancing, excellent horizontal scaling, automatic resource cleanup, and automatic fault tolerance.

Context

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

For the detailed sample code of Tunnel clients, visit Github.

Automatic data processing

Tunnel clients regularly check heartbeats to detect active channels, update status of Channel and ChannelConnect, and initialize, run, and terminate data processing tasks.

1. Initialize the resources of tunnel clients.
   i. Change the state of the tunnel client from Ready to Started.
   ii. Set HeartbeatTimeout and ClientTag in TunnelWorkerConfig to run the ConnectTunnel task and connect Tunnel Service to obtain the ClientId of the current tunnel client.
   iii. Initialize ChannelDialer to create a ChannelConnect task.

      Each ChannelConnect task corresponds to a Channel. ChannelConnect tasks record data consumption checkpoints.
   iv. 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.
   v. Initialize TunnelStateMachine to automatically update the status of the Channel.
2. Regularly check heartbeat messages.

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

ii. Merge the list of channels obtained from Tunnel Service with the local list of channels, and then 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 the WAIT state for a channel that has no ChannelConnect task. If the ChannelConnect task corresponds to a channel in the OPEN state, 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 state of ChannelConnect based on the state of the channel with the same ChannelId. For example, if channels are in the CLOSE state, set the state of corresponding ChannelConnect tasks to CLOSED 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.

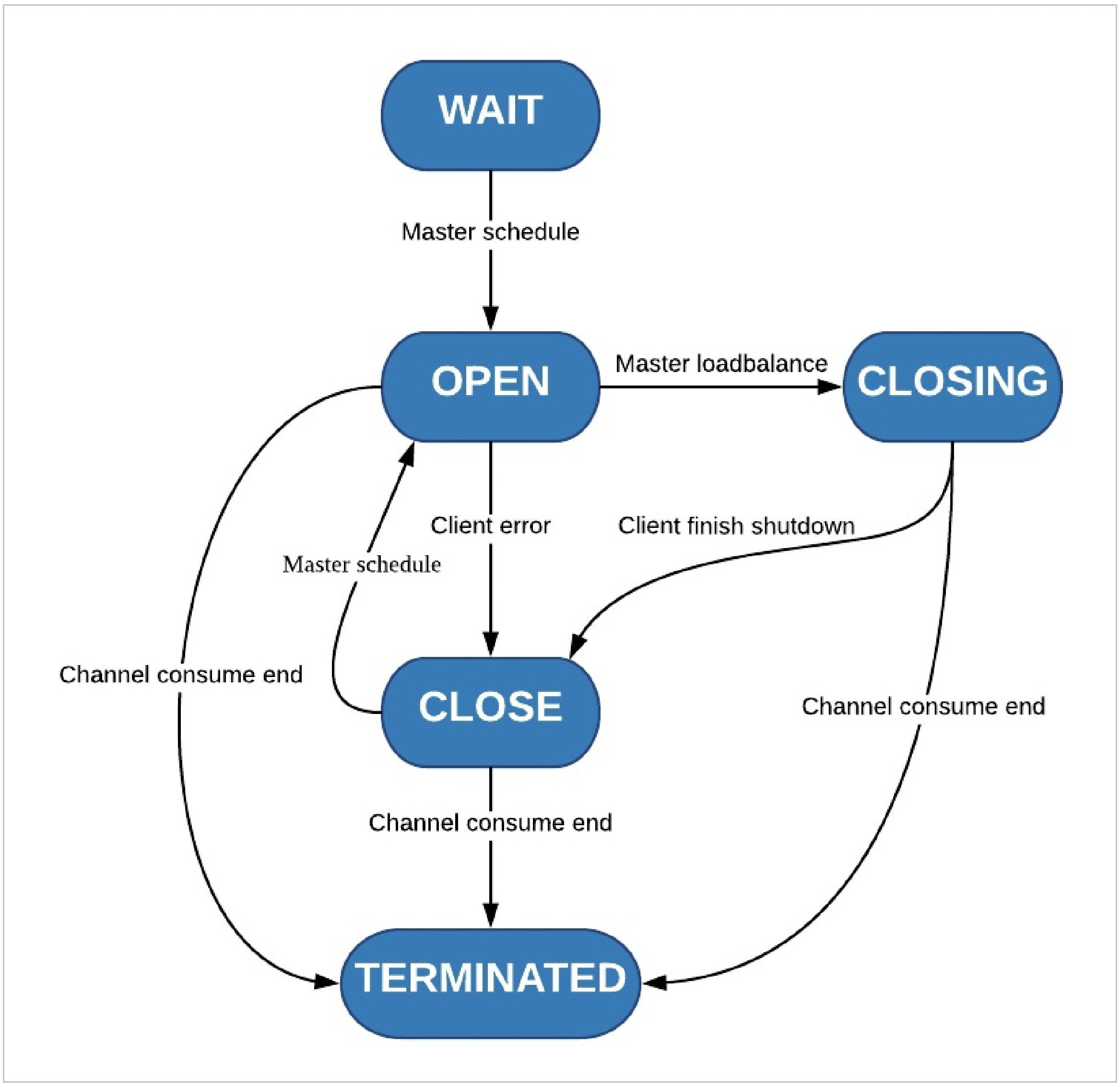
   Tunnel Service and tunnel clients change their status based on heartbeats and channel version updates.

   i. Each channel is initially in the WAIT state.

   ii. The channel for incremental data changes to the OPEN state only when the channel consumption on the parent partition is terminated.

   iii. Tunnel Service allocates the partition in the OPEN state to each tunnel client.

   iv. During load balancing, Tunnel Service and tunnel clients use a scheduling protocol for changing a channel state 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 add tunnel clients to scale out data consumption capabilities. Tunnel clients can run on one or more instances.

Automatic resource cleanup and fault tolerance

- Resource cleanup: 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 non-parametric errors such as heartbeat timeout, occurs on a tunnel client, the system automatically renews connections to continue stable data consumption.

9.4. Quick start

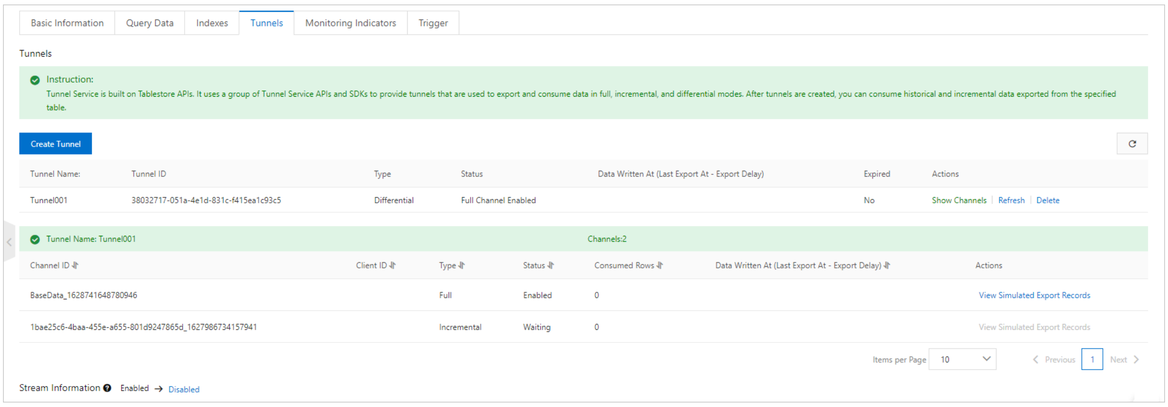
You can use Tunnel Service in the Tablestore console.

Create a tunnel

1. Log on to the Tablestore console.
2. On the **Overview** page, click the name of the target instance or click **Manage Instance** in the Actions column.
3. In the **Tables** section of the **Instance Details** tab, click the name of the target table and click the **Tunnels** tab. You can also click ⋮ in the Actions column and select **Tunnels**.
4. On the **Tunnels** tab, click **Create Tunnel**.
5. 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**. **Incremental** is selected in this topic as an example.

   After the tunnel is created, you can click **Show Channels** in the Actions column to check the data in the tunnel, monitor consumption latency, and check the number of consumed rows in each channel.



Tunnels

Instruction:
Tunnel Service is built on Tablestore APIs. It uses a group of Tunnel Service APIs and SDKs to provide tunnels that are used to export and consume data in full, incremental, and differential modes. After tunnels are created, you can consume historical and incremental data exported from the specified table.

Tunnel Name:	Tunnel ID	Type	Status	Data Written At (Last Export At - Export Delay)	Expired	Actions
Tunnel001	38032717-051a-4e1d-831c-f415ea1c93c5	Differential	Full Channel Enabled		No	Show Channels | Refresh | Delete

Tunnel Name: Tunnel001 Channels:2

Channel ID	Client ID	Type	Status	Consumed Rows	Data Written At (Last Export At - Export Delay)	Actions
BaseData_1628741648780946		Full	Enabled	0		View Simulated Export Records
1bae25c6-4baa-455e-a655-801d9247865d_1627986734157941		Incremental	Waiting	0		View Simulated Export Records

Stream Information Enabled → Disabled

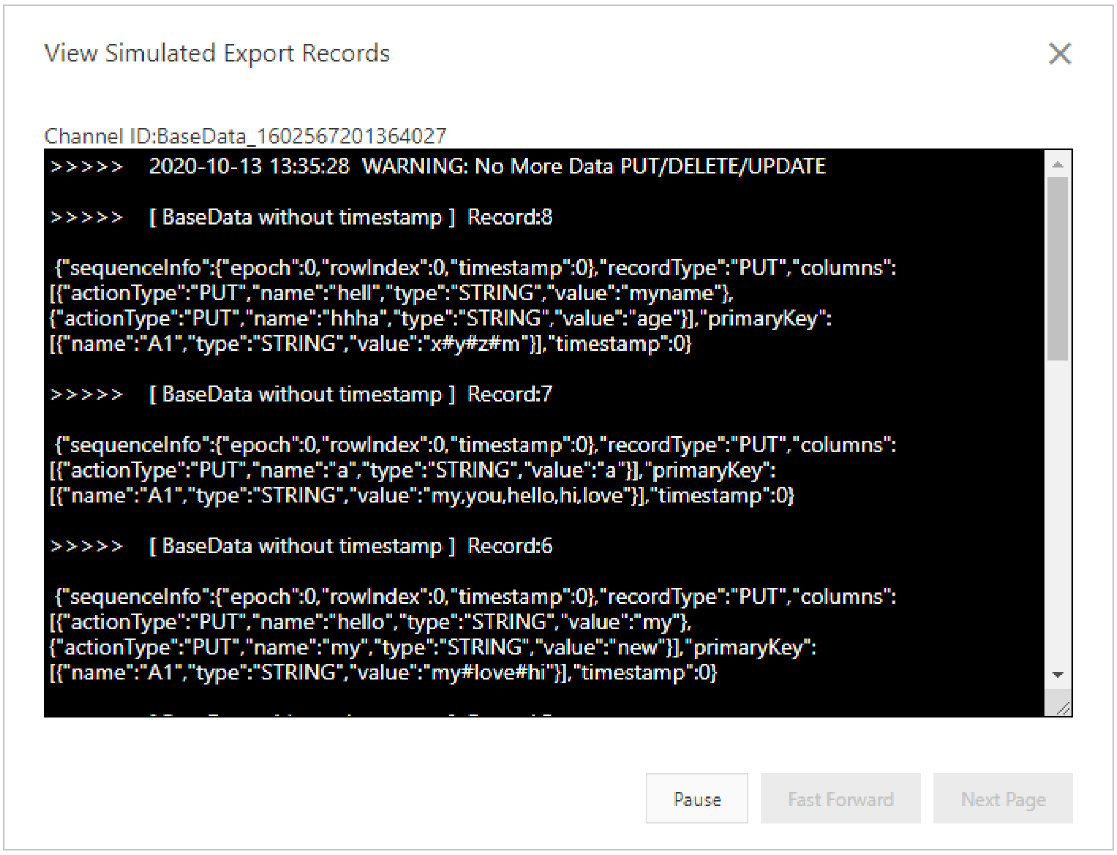
Preview data types in a tunnel

After you create a tunnel, you can simulate data consumption to preview the data types in the tunnel.

1. For more information about how to write data to or delete data from tables in the console, see Read and write data in the console.
2. Preview the data types in a tunnel
   i. On the **Overview** page, click the name of the target instance or click **Manage Instance** in the Actions column.
   ii. In the **Tables** section of the **Instance Details** tab, click the name of the target table and click the **Tunnels** tab. You can also click ⋮ in the Actions column and select **Tunnels**.
   iii. On the **Tunnels** tab, click **Show Channels** in the Actions column.
   iv. Click **View Simulated Export Records** in the Actions column.

v. In the **View Simulated Export Records** dialog box that appears, click **Start**.

The information about consumed data is displayed, as shown in the following figure.



Channel ID:BaseData_1602567201364027

```
>>>>>   2020-10-13 13:35:28  WARNING: No More Data PUT/DELETE/UPDATE

>>>>>   [ BaseData without timestamp ]  Record:8

 {"sequenceInfo":{"epoch":0,"rowIndex":0,"timestamp":0},"recordType":"PUT","columns":
[{"actionType":"PUT","name":"hell","type":"STRING","value":"myname"},
{"actionType":"PUT","name":"hhha","type":"STRING","value":"age"}],"primaryKey":
[{"name":"A1","type":"STRING","value":"x#y#z#m"}],"timestamp":0}

>>>>>   [ BaseData without timestamp ]  Record:7

 {"sequenceInfo":{"epoch":0,"rowIndex":0,"timestamp":0},"recordType":"PUT","columns":
[{"actionType":"PUT","name":"a","type":"STRING","value":"a"}],"primaryKey":
[{"name":"A1","type":"STRING","value":"my,you,hello,hi,love"}],"timestamp":0}

>>>>>   [ BaseData without timestamp ]  Record:6

 {"sequenceInfo":{"epoch":0,"rowIndex":0,"timestamp":0},"recordType":"PUT","columns":
[{"actionType":"PUT","name":"hello","type":"STRING","value":"my"},
{"actionType":"PUT","name":"my","type":"STRING","value":"new"}],"primaryKey":
[{"name":"A1","type":"STRING","value":"my#love#hi"}],"timestamp":0}
```

Enable data consumption for a tunnel

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

```
// Customize the data consumption callback or call the IChannelProcessor operation. Spe
cify the process and shutdown methods.
private static class SimpleProcessor implements IChannelProcessor {
    @Override
    public void process(ProcessRecordsInput 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.getNextToken()));
        try {
            // Simulate the processing of data consumption.
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}
// We recommend that you share the same TunnelWorkerConfig. TunnelWorkerConfig provides
more advanced parameters.
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
// Configure TunnelWorker and start automatic data processing.
TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
try {
    worker.connectAndWorking();
} catch (Exception e) {
    e.printStackTrace();
    worker.shutdown();
    tunnelClient.shutdown();
}
```

View data consumption logs

After data is consumed, you can view the consumption logs of incremental data, including consumption statistics and the latest synchronization time of incremental data in channels. You can also log on to the Tablestore console or call the DescribeTunnel operation to view the consumption latency and the number of consumed rows in each channel.

9.5. SDK usage

Tablestore SDKs allow you to use Tunnel Service. Before you use Tunnel Service, you must familiarize yourself with the usage notes and API operations of Tunnel Service.

Usage notes

- By default, the system starts a thread pool to read and process data based on TunnelWorkerConfig. If you want to start multiple TunnelWorkers on a single server, we recommend that you configure the TunnelWorkers to share the same TunnelWorkerConfig.
- If you create a differential tunnel to consume full and incremental data, the incremental data logs of the tunnel are retained for a maximum of seven days. The specific expiration time of incremental logs

is consistent with that of logs in streams for a data table. If the tunnel does not consume full data within seven days, an OTSTunnelExpired error occurs when the tunnel starts to consume incremental data. As a result, the tunnel cannot consume incremental data. If you estimate that the tunnel cannot consume full data within seven days, submit a ticket or join DingTalk group 23307953 to contact technical support.

- TunnelWorker requires time to warm up for initialization. The heartbeatIntervalInSec parameter in TunnelWorkerConfig specifies the time that is required for TunnelWorker to warm up. You can use the setHeartbeatIntervalInSec method in TunnelWorkerConfig to set this parameter. The default value is 30s. The minimum value is 5s.
- When the mode switches from the full channel to the incremental channel, the full channel is closed and the incremental channel is started. This process requires a period of time for initialization. The heartbeatIntervalInSec parameter specifies the initialization time.
- When the TunnelWorker client is shut down due to an unexpected exit or manual termination, TunnelWorker uses one of the following methods to automatically recycle resources: Release the thread pool. Automatically call the shutdown method that you have registered for the Channel class, and shut down the tunnel.

Operations

Operation	Description
CreateTunnel	Creates a tunnel
ListTunnel	Queries the information about the tunnels in the specified table.
DescribeTunnel	Queries the information about the channels in the specified tunnel.
DeleteTunnel	Deletes a tunnel.

Use Tablestore SDKs

You can use Tablestore SDKs in the following programming languages to implement Tunnel Service:

- Java SDK
- Go SDK

Use Tunnel Service

Tablestore SDK for Java allows you to use Tunnel Service.

1. Initialize a TunnelClient instance.

```
// Set endPoint to the endpoint of the Tablestore instance. Example: https://instance.cn-hangzhou.ots.aliyuncs.com.
// Set accessKeyId to the AccessKey ID and accessKeySecret to the AccessKey secret that you use to access Tablestore.
// Set instanceName to the name of the instance.
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret, instanceName);
```

2. Create a tunnel.

   Create a table for testing or prepare an existing table before you create a tunnel. To create a table for testing, you can use the createTable method in the SyncClient class or go to the Tablestore console.

```
// You can create three types of tunnels by using TunnelType.BaseData, TunnelType.Stream, and TunnelType.BaseAndStream.
// The following code provides an example on how to create a differential tunnel. To create a tunnel of another type, set TunnelType in CreateTunnelRequest to the required type.
final String tableName = "testTable";
final String tunnelName = "testTunnel";
CreateTunnelRequest request = new CreateTunnelRequest(tableName, tunnelName, TunnelType.BaseAndStream);
CreateTunnelResponse resp = tunnelClient.createTunnel(request);
// Use tunnelId to initialize TunnelWorker. You can call the ListTunnel or DescribeTunnel operation to obtain the tunnel ID.
String tunnelId = resp.getTunnelId();
System.out.println("Create Tunnel, Id: " + tunnelId);
```

3. Customize the data consumption callback to start automatic data consumption.

```
// Customize the data consumption callback to implement the IChannelProcessor operation
. Specify the process and shutdown methods.
private static class SimpleProcessor implements IChannelProcessor {
    @Override
    public void process(ProcessRecordsInput input) {
        // ProcessRecordsInput includes the data that you have obtained.
        System.out.println("Default record processor, would print records count");
        System.out.println(
            // NextToken is used to paginate the data of TunnelClient.
            String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
        try {
            // Simulate the processing of data consumption.
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}
// By default, the system starts a thread pool to read and process data based on Tunnel
WorkerConfig.
// If you want to start multiple TunnelWorkers on a single server, we recommend that yo
u configure the TunnelWorkers to share the same TunnelWorkerConfig. TunnelWorkerConfig
contains the larger number of advanced parameters.
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
// Configure TunnelWorker and start automatic data processing.
TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
try {
    worker.connectAndWorking();
} catch (Exception e) {
    e.printStackTrace();
    worker.shutdown();
    tunnelClient.shutdown();
}
```

Configure TunnelWorkerConfig

TunnelWorkerConfig allows you to customize parameters for a TunnelClient instance based on your requirements. The following table describes the parameters.

Item	Parameter	Description
Configure the interval to detect heartbeats and	heartbeatTimeoutInSec	The timeout period to receive heartbeats. Default value: 300. Unit: seconds. When a heartbeat timeout occurs, the tunnel server considers that the current TunnelClient instance is unavailable. The tunnel client must try to reconnect to the tunnel server.

Item	Parameter	Description
detect heartbeats and the timeout period to receive heartbeats	heartbeatIntervalInSec	The interval to detect heartbeats. You can detect heartbeats to detect active channels, update the status of channels, and automatically initialize data processing tasks. Default value: 30. Minimum value: 5. Unit: seconds.
Interval between checkpoints	checkpointIntervalInMillis	The interval between checkpoints when data is consumed. The interval is recorded on the tunnel server. Default value: 5000. Unit: ms. Note • Data to read is stored in different servers. Various errors may occur when you run processes. For example, the server may restart due to environmental factors. The tunnel server regularly records checkpoints after data is processed. A task processes data from the previous checkpoint after the task is restarted. In exceptional conditions, Tunnel Service may sequentially synchronize data once or multiple times. If some data is reprocessed, check the business processing logic. • To prevent data being reprocessed when errors occur, record more checkpoints. However, an excessive number of checkpoints may compromise the system throughput. We recommend that you record the checkpoints based on your business requirements.
The client tag	clientTag	The custom client tag that is used to generate a tunnel client ID. You can customize this parameter to uniquely identify TunnelWorkers.
The custom callback to process data	channelProcessor	The callback that you register to process data, including the process and shutdown methods.
	readRecordsExecutor	The thread pool to read data. If you do not have special requirements, use the default configuration.

Item	Parameter	Description
The configuration of the thread pool to read and process data	processRecordsExecutor	The thread pool to process data. If you do not have special requirements, use the default configuration. Note - When you customize the thread pool, we recommend that you set the number of threads to the number of the channels in the tunnel. This way, each channel can be quickly allocated with computing resources such as CPU. - Tablestore performs the following operations for the default configurations of the pool to ensure throughput: ○ Allocate 32 core threads in advance to guarantee real-time throughput when a small amount of data or a small number of channels exists. ○ Reduce the queue length when a large amount of data must be processed or when a large number of channels exist. This way, the policy is triggered to create a thread in the pool and quickly allocate more computing resources. ○ We recommend that you set the thread keep-alive time to 60s. If the amount of data that must be processed is reduced, you can recycle thread resources.
Memory control	maxChannelParallel	The concurrency level of channels to read and process data for memory control. The default value is -1, which indicates that the concurrency level is unlimited. Note This configuration applies only to Tablestore SDK for Java V5.10.0 and later.

Item	Parameter	Description
Maximum backoff time	maxRetryIntervalInMillis	The reference value to calculate the maximum backoff time for the tunnel. The maximum backoff time is set to a random value around the reference value. The range of valid values for the maximum backoff time is computed by using the formula: 0.75 × maxRetryIntervalInMillis to 1.25 × maxRetryIntervalInMillis. Default value: 2000 ms. Minimum value: 200 ms. Note • This configuration applies only to Tablestore SDK for Java V5.4.0 and later. • If the amount of data that must be processed is smaller than 900 KB or 500 pieces for each export, the tunnel client uses exponential backoff for the tunnel until the maximum backoff time is reached.

Appendix: complete code

```
import com.alicloud.openservices.tablestore.TunnelClient;
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelRequest;
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelResponse;
import com.alicloud.openservices.tablestore.model.tunnel.TunnelType;
import com.alicloud.openservices.tablestore.tunnel.worker.IChannelProcessor;
import com.alicloud.openservices.tablestore.tunnel.worker.ProcessRecordsInput;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorker;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorkerConfig;
public class TunnelQuickStart {
    private static class SimpleProcessor implements IChannelProcessor {
        @Override
        public void process(ProcessRecordsInput input) {
            System.out.println("Default record processor, would print records count");
            System.out.println(
                // NextToken is used to paginate the data of TunnelClient.
                String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
            try {
                // Simulate the processing of data consumption.
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void shutdown() {
            System.out.println("Mock shutdown");
        }
    }
    public static void main(String[] args) throws Exception {
        //1.Initialize a TunnelClient instance.
        final String endPoint = "";
        final String accessKeyId = "";
        final String accessKeySecret = "";
        final String instanceName = "";
        TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret
, instanceName);
        //2.Create a tunnel. You must create a table before you create a tunnel. To create
a table, you must use the createTable method in SyncClient or go to the Tablestore console.

        final String tableName = "testTable";
        final String tunnelName = "testTunnel";
        CreateTunnelRequest request = new CreateTunnelRequest(tableName, tunnelName, Tunnel
Type.BaseAndStream);
        CreateTunnelResponse resp = tunnelClient.createTunnel(request);
        // Use tunnelId to initialize TunnelWorker. You can call the ListTunnel or Describe
Tunnel operation to obtain the tunnel ID.
        String tunnelId = resp.getTunnelId();
        System.out.println("Create Tunnel, Id: " + tunnelId);
        //3.Customize the data consumption callback to start automatic data consumption.
        // TunnelWorkerConfig contains the larger number of advanced parameters.
        TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
        TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
        try {
            worker.connectAndWorking();
        } catch (Exception e) {
            e.printStackTrace();
            worker.shutdown();
            tunnelClient.shutdown();
        }
    }
}
```

9.6. Troubleshooting

This topic describes the format of error messages and error codes returned by Tunnel Service.

After Tunnel Service receives an abnormal request, Tunnel Service returns an HTTP status code and an error message in the Protobuf format.

Error message format

A error message returned by Tunnel Service is in the following Protobuf format:

```
message Error {
    required string code = 1;
    optional string message = 2;
    optional string tunnel_id = 3;
}
```

Error codes

When you use Tunnel Service SDKs, you need only to handle the error codes whose processing logic is "Return an error message". The SDKs automatically process other error codes and retry the requests. We recommend that you handle an error based on the processing logic of the error code.

HTTP status code	Error code	Description	Processing logic
400	OTSParameterInvalid	The error message returned because an parameter in the API request is incorrect or the requested data table does not exist.	Return an error message.
400	OTSTunnelExpired	The error message returned because the log data in an incremental or a differential tunnel expires.	Return an error message.
403	OTSPermissionDenied	The error message returned because you are not authorized to access the specified resource.	Return an error message.
409	OTSTunnelExist	The error message returned because the tunnel that you want to create already exists on the server.	Return an error message.
400	OTSSequenceNumberNotMatch	The error message returned because the serial numbers of checkpoints are inconsistent between the client and server. This error can occur when the serial numbers of checkpoints on the client are smaller than those on the server or channels compete with each other.	Use the checkpoint API to obtain the serial numbers of checkpoints again.

HTTP status code	Error code	Description	Processing logic
410	OTSResourceGone	The error message returned because the request sent to obtain the heartbeat information about the Tunnel Service client times out.	Use the tunnel ID to reconnect to Tunnel Service.
503	OTSTunnelServerUnavailable	The error message returned because an internal server error occurs.	Use exponential backoff to retry the request.

9.7. Incremental synchronization performance white paper

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

Test environment

- Tablestore instance
  - Type: high-performance instance
  - Region: China (Hangzhou)
  - Address: a private IP address, which prevents interferences caused by unknown network issues.
- Test server
  - Type: Alibaba Cloud ECS
  - Region: China (Hangzhou)
  - Model: the ecs.mn4.4xlarge shared balanced instance type
  - 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 Tablestore can write data to multiple rows simultaneously by using Tablestore SDK for Java to call the BatchWriteRow operation.

- Prepartitioning tool

  The stress testing tool of Tablestore can automatically create and prepartition tables based on the configured table names and the number of partitions.

- Rate statistics tool

  Tablestore SDK for Java can collect statistics for 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.

  Examples

```
private static final Gson GSON = new Gson();
    private static final int CAL_INTERVAL_MILLIS = 5000;
    static class PerfProcessor implements IChannelProcessor {
        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(ProcessRecordsInput input) {
            counter.addAndGet(input.getRecords().size());
            allCount.addAndGet(input.getRecords().size());
            if (System.currentTimeMillis() - latestTs.get() > CAL_INTERVAL_MILLIS) {
                synchronized (PerfProcessor.class) {
                    if (System.currentTimeMillis() - latestTs.get() > CAL_INTERVAL_MILLIS
) {
                        long seconds = TimeUnit.MILLISECONDS.toSeconds(System.currentTime
Millis() - latestTs.get());
                        PerfElement element = new PerfElement(System.currentTimeMillis(),
counter.get() / seconds, allCount.get());
                        System.out.println(GSON.toJson(element));
                        counter.set(0);
                        latestTs.set(System.currentTimeMillis());
                    }
                }
            }
        }
        @Override
        public void shutdown() {
            System.out.println("Mock shutdown");
        }
    }
```

Test plan

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

> **Note**
>
> - The number of partitions increases with the data volume. To create partitions in advance, contact Tablestore technical support.
> - Data processing tasks are automatically implemented in Tunnel Service. The tunnel client can be started by using the same tunnel ID. For more information, see Tunnel clients.

- 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 Tablestore instance.

- Test procedure

  i. Create and pre-split a table for each test scenario.

  ii. Create a tunnel for incremental synchronization.

  iii. Use the stress testing tool to write incremental data.

  iv. Use the rate statistics tool to measure the queries per second (QPS) in real time, and check the consumption of system resources such as CPU and memory.

  v. Check the total bandwidth consumed during incremental synchronization.

- Test data description

  Sample 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 ensures that stress testing data is evenly written to each partition.

Test indicators

The following indicators are used in the test:

- QPS (row): the number of rows synchronized per second.
- Average latency (ms per 1,000 rows): the amount of time in milliseconds required to synchronize 1,000 rows.
- CPU (cores): the total number of single-core CPUs used to synchronize data.
- Memory (GB): the total physical memory used to synchronize data.
- Bandwidth (Mbit/s): the total bandwidth used to synchronize data.

> **Note** The 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 the test details.

- QPS and latency

  The number of rows synchronized per second and the amount of time required to synchronize 1,000 rows in each scenario. 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, which results in QPS of only 570,000. For more information, see the test details. The average QPS in the double-server 64-partition synchronization scenario reaches 780,000 rows, almost twice as much as the 420,000 QPS achieved in the single-server 32-partition synchronization scenario. In the double-server 128-partition synchronization scenario, the QPS reaches 1,000,000 rows.

- System resource consumption

  The CPU usage increases linearly with the number of partitions.

  In the single-server single-partition synchronization scenario, 0.25 single-core CPUs are used. In the double-server 128-partition synchronization scenario, only 10.2 single-core CPUs are used when the QPS reaches 1,000,000 rows. The memory usage increases linearly with the number of partitions when the number of partitions is smaller than 32. When more partitions (32 or 64 partitions) need to be processed, the memory usage remains stable around 5.3 GB on each server.

- Total bandwidth consumption

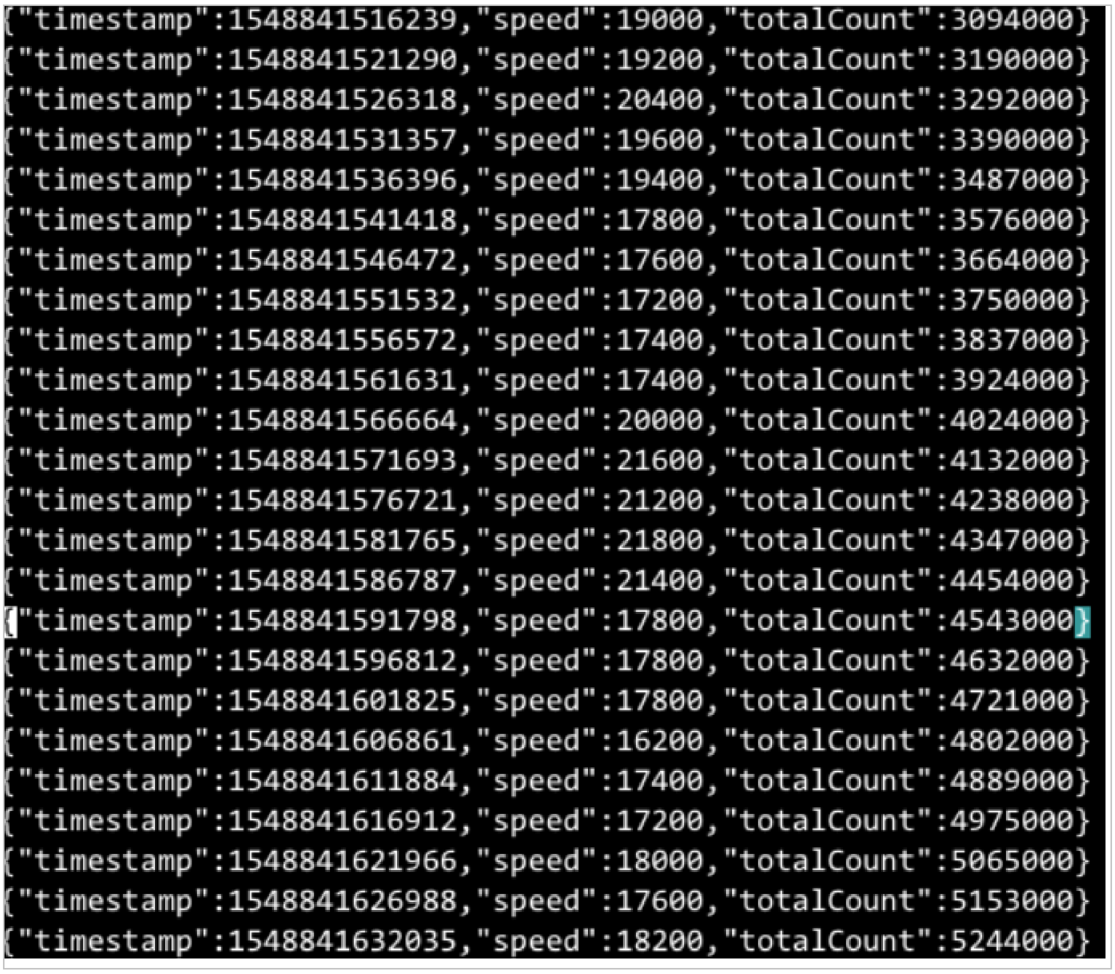
  The consumed bandwidth increases linearly with the number of partitions.

  In the single-server 64-partition synchronization scenario, a total bandwidth of 125 Mbit/s is consumed, which is the maximum rate supported by the gigabit NIC. In the double-server 64-partition synchronization scenario, a bandwidth of 169 Mbit/s is consumed, 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 on 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.

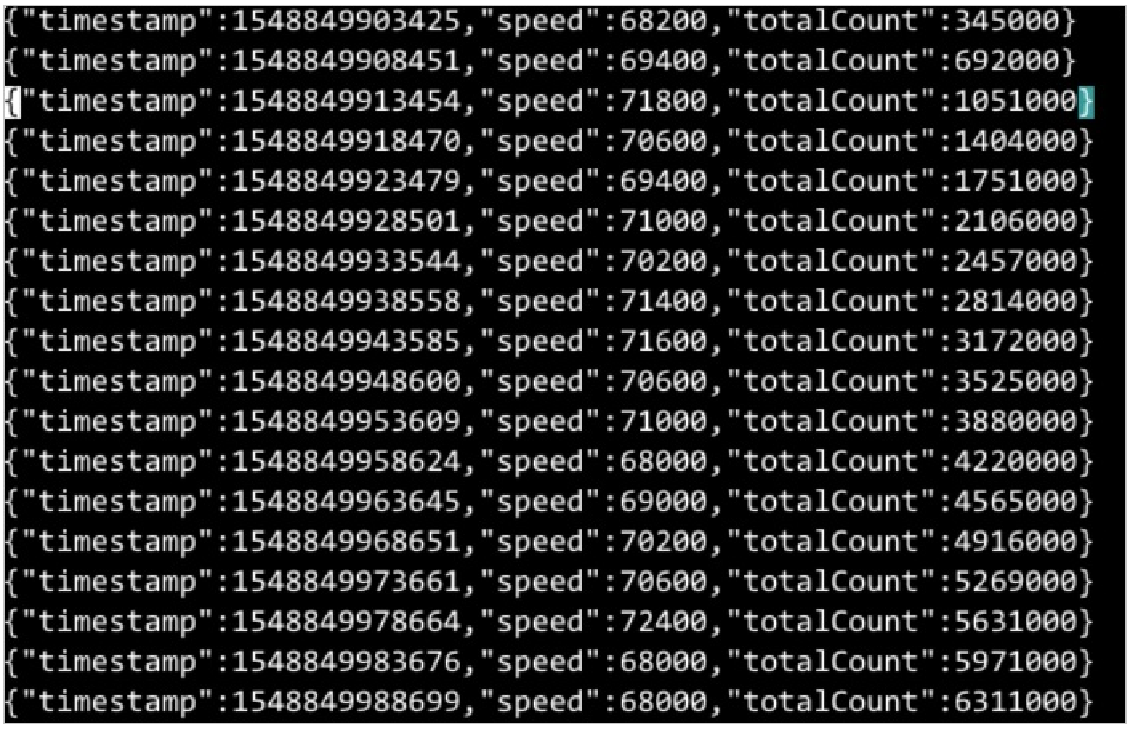


```
{"timestamp":1548841516239,"speed":19000,"totalCount":3094000}
{"timestamp":1548841521290,"speed":19200,"totalCount":3190000}
{"timestamp":1548841526318,"speed":20400,"totalCount":3292000}
{"timestamp":1548841531357,"speed":19600,"totalCount":3390000}
{"timestamp":1548841536396,"speed":19400,"totalCount":3487000}
{"timestamp":1548841541418,"speed":17800,"totalCount":3576000}
{"timestamp":1548841546472,"speed":17600,"totalCount":3664000}
{"timestamp":1548841551532,"speed":17200,"totalCount":3750000}
{"timestamp":1548841556572,"speed":17400,"totalCount":3837000}
{"timestamp":1548841561631,"speed":17400,"totalCount":3924000}
{"timestamp":1548841566664,"speed":20000,"totalCount":4024000}
{"timestamp":1548841571693,"speed":21600,"totalCount":4132000}
{"timestamp":1548841576721,"speed":21200,"totalCount":4238000}
{"timestamp":1548841581765,"speed":21800,"totalCount":4347000}
{"timestamp":1548841586787,"speed":21400,"totalCount":4454000}
{"timestamp":1548841591798,"speed":17800,"totalCount":4543000}
{"timestamp":1548841596812,"speed":17800,"totalCount":4632000}
{"timestamp":1548841601825,"speed":17800,"totalCount":4721000}
{"timestamp":1548841606861,"speed":16200,"totalCount":4802000}
{"timestamp":1548841611884,"speed":17400,"totalCount":4889000}
{"timestamp":1548841616912,"speed":17200,"totalCount":4975000}
{"timestamp":1548841621966,"speed":18000,"totalCount":5065000}
{"timestamp":1548841626988,"speed":17600,"totalCount":5153000}
{"timestamp":1548841632035,"speed":18200,"totalCount":5244000}
```

- CPU utilization: approximately 25% of a single-core CPU.
- Memory usage: approximately 0.4% of the total physical memory, or 0.256 GB. (Each test server provides 64 GB of physical memory.)
- Bandwidth consumption: approximately 4,000 KB/s.

- Single-server 4-partition synchronization: 70,000 QPS.
  - Tested at: 20:00 on 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.

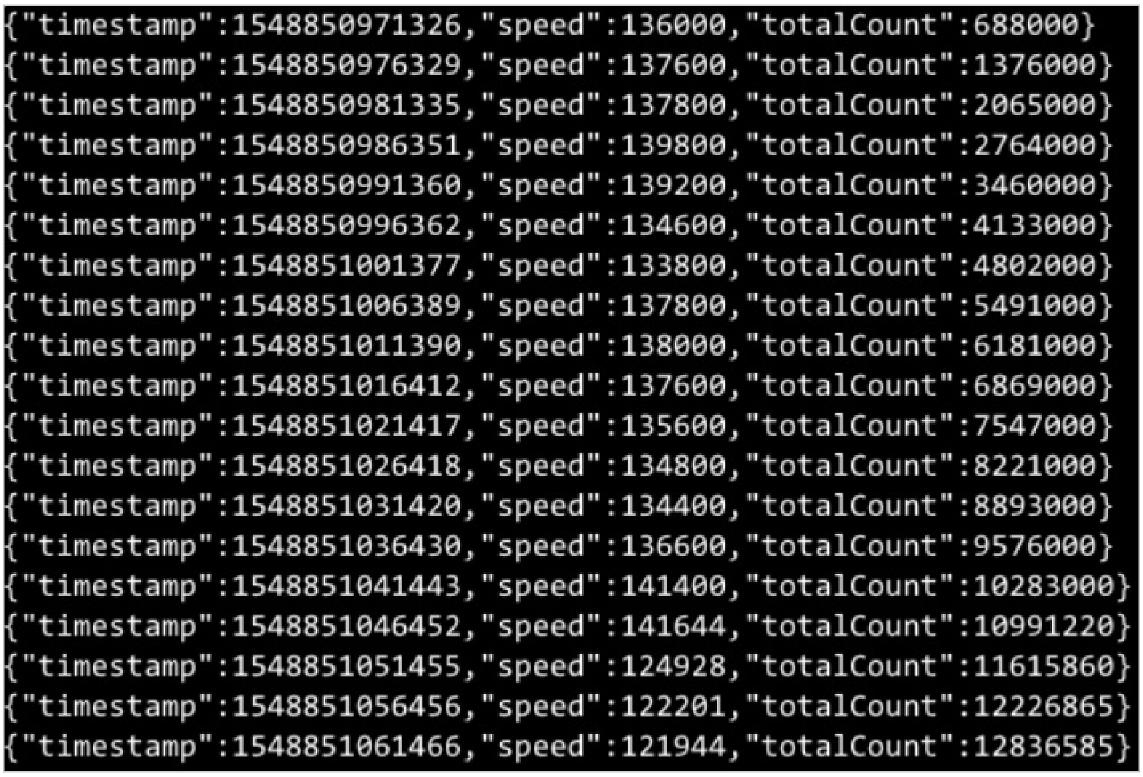


```
{"timestamp":1548849903425,"speed":68200,"totalCount":345000}
{"timestamp":1548849908451,"speed":69400,"totalCount":692000}
{"timestamp":1548849913454,"speed":71800,"totalCount":1051000}
{"timestamp":1548849918470,"speed":70600,"totalCount":1404000}
{"timestamp":1548849923479,"speed":69400,"totalCount":1751000}
{"timestamp":1548849928501,"speed":71000,"totalCount":2106000}
{"timestamp":1548849933544,"speed":70200,"totalCount":2457000}
{"timestamp":1548849938558,"speed":71400,"totalCount":2814000}
{"timestamp":1548849943585,"speed":71600,"totalCount":3172000}
{"timestamp":1548849948600,"speed":70600,"totalCount":3525000}
{"timestamp":1548849953609,"speed":71000,"totalCount":3880000}
{"timestamp":1548849958624,"speed":68000,"totalCount":4220000}
{"timestamp":1548849963645,"speed":69000,"totalCount":4565000}
{"timestamp":1548849968651,"speed":70200,"totalCount":4916000}
{"timestamp":1548849973661,"speed":70600,"totalCount":5269000}
{"timestamp":1548849978664,"speed":72400,"totalCount":5631000}
{"timestamp":1548849983676,"speed":68000,"totalCount":5971000}
{"timestamp":1548849988699,"speed":68000,"totalCount":6311000}
```

   - CPU utilizatuin: approximately 70% of a single-core CPU.
   - Memory usage: approximately 1.9% of the total physical memory, or 1.1 GB. (Each test server provides 64 GB of physical memory.)
   - Bandwidth consumption: approximately 13 Mbit/s.

- Single-server 8-partition synchronization: 130,000 QPS.
   - Tested at: 20:20 on 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.

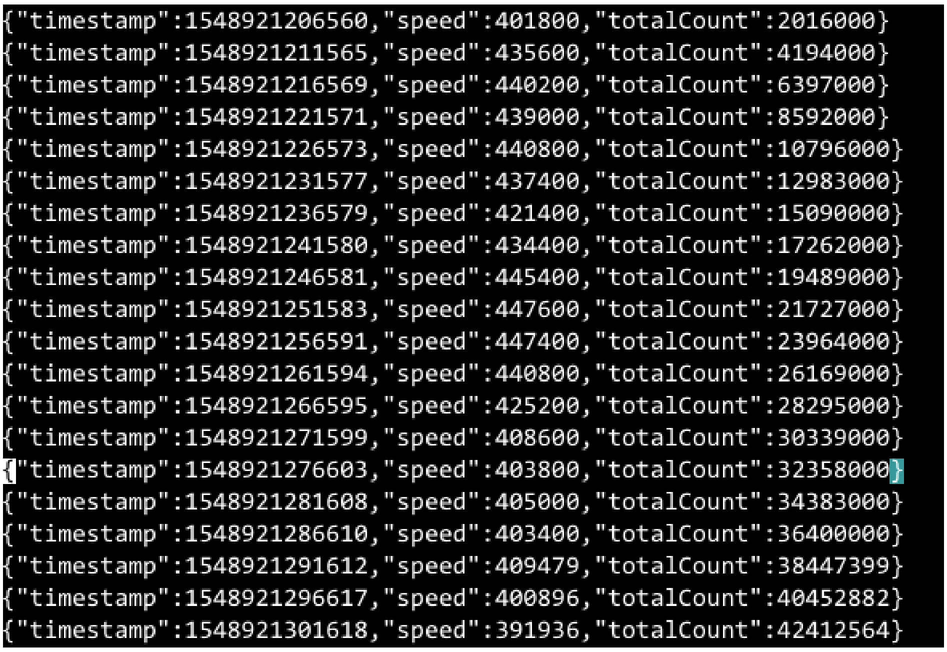


```
{"timestamp":1548850971326,"speed":136000,"totalCount":688000}
{"timestamp":1548850976329,"speed":137600,"totalCount":1376000}
{"timestamp":1548850981335,"speed":137800,"totalCount":2065000}
{"timestamp":1548850986351,"speed":139800,"totalCount":2764000}
{"timestamp":1548850991360,"speed":139200,"totalCount":3460000}
{"timestamp":1548850996362,"speed":134600,"totalCount":4133000}
{"timestamp":1548851001377,"speed":133800,"totalCount":4802000}
{"timestamp":1548851006389,"speed":137800,"totalCount":5491000}
{"timestamp":1548851011390,"speed":138000,"totalCount":6181000}
{"timestamp":1548851016412,"speed":137600,"totalCount":6869000}
{"timestamp":1548851021417,"speed":135600,"totalCount":7547000}
{"timestamp":1548851026418,"speed":134800,"totalCount":8221000}
{"timestamp":1548851031420,"speed":134400,"totalCount":8893000}
{"timestamp":1548851036430,"speed":136600,"totalCount":9576000}
{"timestamp":1548851041443,"speed":141400,"totalCount":10283000}
{"timestamp":1548851046452,"speed":141644,"totalCount":10991220}
{"timestamp":1548851051455,"speed":124928,"totalCount":11615860}
{"timestamp":1548851056456,"speed":122201,"totalCount":12226865}
{"timestamp":1548851061466,"speed":121944,"totalCount":12836585}
```

  - CPU utilization: approximately 120% of a single-core CPU.
  - Memory usage: approximately 4.1% of the total physical memory, or 2.62 GB. (Each test server provides 64 GB of physical memory.)
  - Bandwidth consumption: approximately 27 Mbit/s.

- Single-server 32-partition synchronization: 420,000 QPS.
  - Tested at: 15:50 on January 31, 2019.
  - QPS: steady at approximately 420,000 rows per second, with a peak rate of 447,600 rows per second.

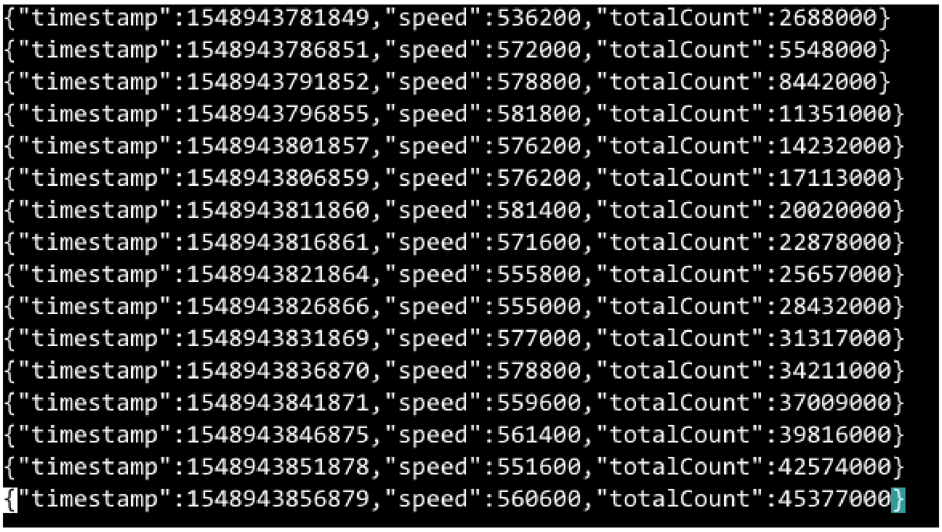
  - Latency: 2.38 ms per 1,000 rows.



```
{"timestamp":1548921206560,"speed":401800,"totalCount":2016000}
{"timestamp":1548921211565,"speed":435600,"totalCount":4194000}
{"timestamp":1548921216569,"speed":440200,"totalCount":6397000}
{"timestamp":1548921221571,"speed":439000,"totalCount":8592000}
{"timestamp":1548921226573,"speed":440800,"totalCount":10796000}
{"timestamp":1548921231577,"speed":437400,"totalCount":12983000}
{"timestamp":1548921236579,"speed":421400,"totalCount":15090000}
{"timestamp":1548921241580,"speed":434400,"totalCount":17262000}
{"timestamp":1548921246581,"speed":445400,"totalCount":19489000}
{"timestamp":1548921251583,"speed":447600,"totalCount":21727000}
{"timestamp":1548921256591,"speed":447400,"totalCount":23964000}
{"timestamp":1548921261594,"speed":440800,"totalCount":26169000}
{"timestamp":1548921266595,"speed":425200,"totalCount":28295000}
{"timestamp":1548921271599,"speed":408600,"totalCount":30339000}
{"timestamp":1548921276603,"speed":403800,"totalCount":32358000}
{"timestamp":1548921281608,"speed":405000,"totalCount":34383000}
{"timestamp":1548921286610,"speed":403400,"totalCount":36400000}
{"timestamp":1548921291612,"speed":409479,"totalCount":38447399}
{"timestamp":1548921296617,"speed":400896,"totalCount":40452882}
{"timestamp":1548921301618,"speed":391936,"totalCount":42412564}
```

  - CPU utilzation: approximately 450% of a single-core CPU.
  - Memory usage: approximately 8.2% of the total physical memory, or 5.25 GB. (Each test server provides 64 GB of physical memory.)
  - Bandwidth consumption: approximately 86 Mbit/s.

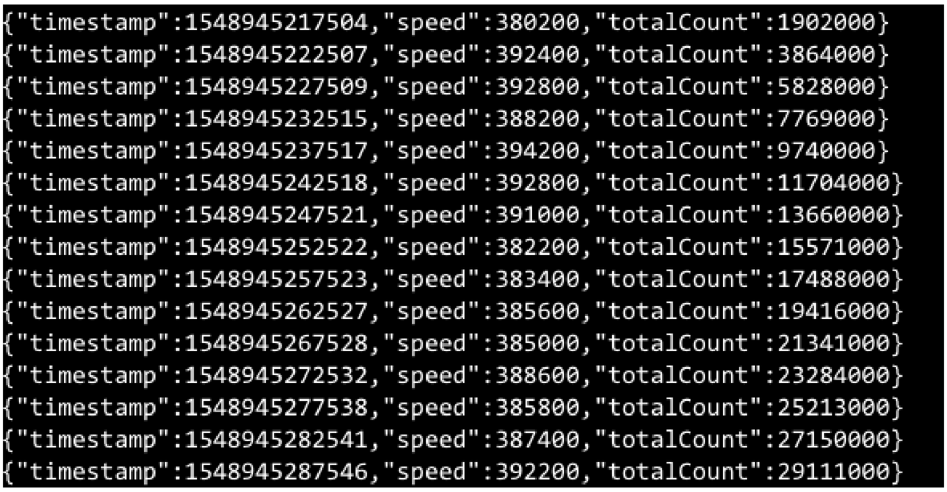
- Single-server 64-partition synchronization: 570,000 QPS, with the gigabit NIC working at its full capacity.
  - Tested at: 22:10 on January 31, 2019.
  - QPS: steady at approximately 570,000 rows per second, with a peak rate of 581,400 rows per second.
  - Latency: approximately 1.75 ms per 1,000 rows.



```
{"timestamp":1548943781849,"speed":536200,"totalCount":2688000}
{"timestamp":1548943786851,"speed":572000,"totalCount":5548000}
{"timestamp":1548943791852,"speed":578800,"totalCount":8442000}
{"timestamp":1548943796855,"speed":581800,"totalCount":11351000}
{"timestamp":1548943801857,"speed":576200,"totalCount":14232000}
{"timestamp":1548943806859,"speed":576200,"totalCount":17113000}
{"timestamp":1548943811860,"speed":581400,"totalCount":20020000}
{"timestamp":1548943816861,"speed":571600,"totalCount":22878000}
{"timestamp":1548943821864,"speed":555800,"totalCount":25657000}
{"timestamp":1548943826866,"speed":555000,"totalCount":28432000}
{"timestamp":1548943831869,"speed":577000,"totalCount":31317000}
{"timestamp":1548943836870,"speed":578800,"totalCount":34211000}
{"timestamp":1548943841871,"speed":559600,"totalCount":37009000}
{"timestamp":1548943846875,"speed":561400,"totalCount":39816000}
{"timestamp":1548943851878,"speed":551600,"totalCount":42574000}
{"timestamp":1548943856879,"speed":560600,"totalCount":45377000}
```

  - CPU utilization: approximately 640% of a single-core CPU.
  - Memory usage: approximately 8.4% of the total physical memory, or 5.376 GB.
  - Bandwidth consumption: approximately 125 Mbit/s, which is the maximum rate of the gigabit NIC.

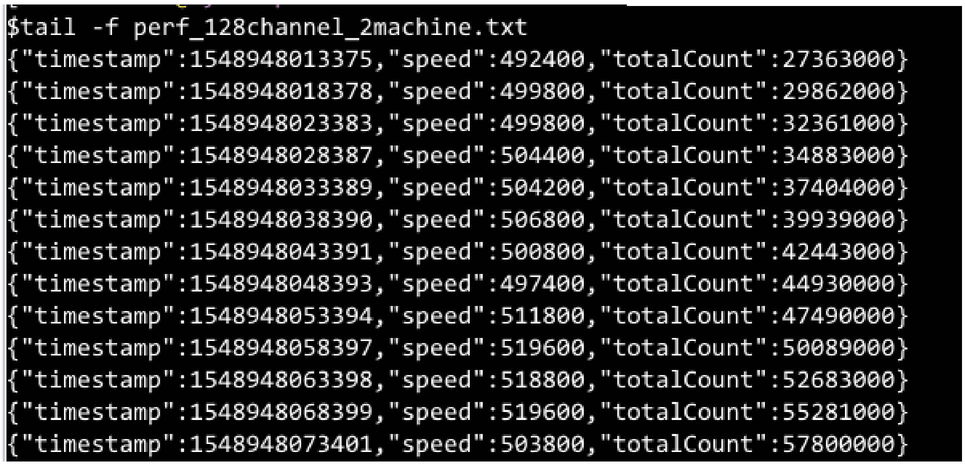
- Double-server 64-partition synchronization: 780,000 QPS.
    - Tested at: 22:30 on January 31, 2019.
    - QPS: steady at approximately 390,000 rows per second on each server and 780,000 rows per second on both servers.
    - Latency: approximately 1.28 ms per 1,000 rows.



```
{"timestamp":1548945217504,"speed":380200,"totalCount":1902000}
{"timestamp":1548945222507,"speed":392400,"totalCount":3864000}
{"timestamp":1548945227509,"speed":392800,"totalCount":5828000}
{"timestamp":1548945232515,"speed":388200,"totalCount":7769000}
{"timestamp":1548945237517,"speed":394200,"totalCount":9740000}
{"timestamp":1548945242518,"speed":392800,"totalCount":11704000}
{"timestamp":1548945247521,"speed":391000,"totalCount":13660000}
{"timestamp":1548945252522,"speed":382200,"totalCount":15571000}
{"timestamp":1548945257523,"speed":383400,"totalCount":17488000}
{"timestamp":1548945262527,"speed":385600,"totalCount":19416000}
{"timestamp":1548945267528,"speed":385000,"totalCount":21341000}
{"timestamp":1548945272532,"speed":388600,"totalCount":23284000}
{"timestamp":1548945277538,"speed":385800,"totalCount":25213000}
{"timestamp":1548945282541,"speed":387400,"totalCount":27150000}
{"timestamp":1548945287546,"speed":392200,"totalCount":29111000}
```

    - CPU utilization: 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, or 10.5 GB.
    - Bandwidth consumption: approximately 169 Mbit/s. This indicates that bandwidth becomes a bottleneck when the number of partitions reaches 64 in single-server scenarios.

- Double-server 128-partition synchronization: 1,000,000 QPS, when both gigabit NICs are working at their full capacities.
    - Tested at: 23:20 on 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.



```
$tail -f perf_128channel_2machine.txt
{"timestamp":1548948013375,"speed":492400,"totalCount":27363000}
{"timestamp":1548948018378,"speed":499800,"totalCount":29862000}
{"timestamp":1548948023383,"speed":499800,"totalCount":32361000}
{"timestamp":1548948028387,"speed":504400,"totalCount":34883000}
{"timestamp":1548948033389,"speed":504200,"totalCount":37404000}
{"timestamp":1548948038390,"speed":506800,"totalCount":39939000}
{"timestamp":1548948043391,"speed":500800,"totalCount":42443000}
{"timestamp":1548948048393,"speed":497400,"totalCount":44930000}
{"timestamp":1548948053394,"speed":511800,"totalCount":47490000}
{"timestamp":1548948058397,"speed":519600,"totalCount":50089000}
{"timestamp":1548948063398,"speed":518800,"totalCount":52683000}
{"timestamp":1548948068399,"speed":519600,"totalCount":55281000}
{"timestamp":1548948073401,"speed":503800,"totalCount":57800000}
```

    - 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 for each server, or 10.5 GB.

- Bandwidth consumption: approximately 220 Mbit/s.

Summary

Based on the performance test for incremental synchronization, the QPS for tables with a single or a few partitions is 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 of Tunnel Service increases linearly until a system bottleneck (such as bandwidth in the 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. The test validates the excellent horizontal scaling performance of Tunnel Service.

10.Data Delivery

10.1. Overview

Tablestore uses data delivery to deliver full or incremental data to Object Storage Service (OSS) that is used as a data lake in real time. This feature enables Tablestore to store historical data in OSS at lower costs while Tablestore implements offline or quasi-real-time analysis of larger amounts of data.

Scenarios

You can use data delivery to address needs of the following scenarios:

- Tiered storage of cold and hot data

  Data delivery uses the time to live (TTL) feature of Tablestore to store full data in OSS at low costs. Tablestore allows you to query and analyze hot data with low latencies.

- Full data backup

  You can use data delivery to deliver data of a whole table to an OSS bucket for backup and archiving.

- Large-scale data analysis in real time

  You can use data delivery to deliver incremental data from Tablestore to OSS in real time (every 2 minutes). Delivered data is partitioned based on the system time and stored in the Parquet format. You can use OSS high-speed bandwidth for reading and optimization of scanning for Parquet data to implement efficient real-time data analysis.

- Accelerated analysis by using SQL statements

  When search indexes are not created for Tablestore data, and the query conditions exclude primary key column-based filter conditions, you can use data delivery to synchronize data to OSS. Then, use DLA and OSS data scanning to accelerate SQL-based analysis.

Features

Data delivery has the following features:

- Data delivery obtains full and incremental data of Tablestore. When the amount of data reaches the predetermined size or after the data is delivered for more than two minutes, the data is stored in OSS.
- Data delivery allows you to deliver data in the following modes: incremental, full, and differential. All delivered data is stored in the Parquet format.
- Data delivery supports the monitoring of the time when data delivery is complete. Data delivery provides the DescribeDeliveryTask operation to return the time when data delivery is complete.

Benefits

- Ease of use

  To deliver data from Tablestore to OSS, you need only to complete simple configurations in the console. Synchronization tasks run and the throughput capacity is scaled based on the load while monitoring and O&M are not required. However, service-level agreements (SLAs) are guaranteed.

- A complete set of data delivery modes

Data delivery modes of full, incremental, and differential are provided. When the incremental mode is set, delivery tasks implement quasi-real-time delivery of data, obtain the latest data, cache the data, and write the data to OSS after 2 minutes.

- Seamless integration with the computational ecology

  Data delivery is compatible with open source ecology standards and the naming conventions followed by Hive. Delivered data is stored in the Parquet format. To analyze the data in an external table after data is delivered to an OSS bucket, use E-MapReduce.

- Tiered storage and access experience

  After data is delivered to OSS, you can access different data such as data in tables and index tables and data delivered to OSS. This way, the analysis requirements of different scenarios are met.

10.2. Quick start

You can create a delivery task in the Tablestore console to deliver data from Tablestore to an OSS bucket.

Prerequisites

OSS is activated. A bucket is created in the region where a Tablestore instance is located. For more information, see Activate OSS.

> ⓘ Note
>
> Data delivery allows you to deliver data from a Tablestore instance to an OSS bucket within the same region. To deliver data to another warehouse such as MaxCompute,submit a ticket.

Usage notes

- Data delivery is available in the China (Hangzhou), China (Shanghai), China (Beijing), and China (Zhangjiakou) regions.
- The delete operation on Tablestore data is ignored when the data is delivered. Tablestore data delivered to OSS is not deleted when you perform a delete operation on the data.
- It takes at most one minute for initialization when you create a delivery task.
- Latencies are within 3 minutes when data is written at a steady rate. P99 is within 10 minutes when data is synchronized.

> ⓘ Note
>
> P99 indicates the average latency of the slowest 1% of requests over the last 10 seconds.

Create delivery tasks

1. Log on to the Tablestore console.
2. Select a region. Click an instance name or click **Manage Instance** in the **Actions** column corresponding to the instance.
3. On the **Manage Instance** page, click the **Deliver Data to OSS** tab.
4. (Optional) Create the AliyunServiceRoleForOTSDataDelivery role.

When you configure data delivery for the first time, you must create the AliyunServiceRoleForOTSDataDelivery role that is used to authorize Tablestore to write data to an OSS bucket. For more information about specific operations, see AliyunServiceRoleForOTSDataDelivery role.

> Note
>
> For more information about service-linked roles, see AliyunServiceRoleForOTSDataDelivery role.

i. On the **Deliver Data to OSS** tab, click **Role for Delivery Service**.

ii. In the **Role Details** message, view related information. Click **OK**.

5. Create a delivery task.

i. On the **Deliver Data to OSS** tab, click **Create Task**.

ii. In the **Create Task** dialog box, configure the following parameters.

Parameter	Description
Task Name	The name of the task. The name must be 3 to 16 characters in length can contain only lowercase letters, digits, and hyphens (-). It must start and end with a lowercase letter or digit.
Destination Region	The region where the Tablestore instance and OSS bucket are located.
Source Table	The name of the Tablestore table.
Destination Bucket	The name of the OSS bucket. Notice The region where the bucket is located must be the same as that of the Tablestore instance.

Parameter	Description
Destination Prefix	The prefix of the folder in the bucket. Data is delivered from Tablestore to the folder. The path of the destination folder supports the following time variables: $yyyy, $MM, $dd, $HH, and $mm. For more information, see Partition data by time. - When the path uses time variables for delivery, OSS folders are dynamically generated based on the time when data is written. This way, data is partitioned based on the naming conventions followed when Hive partitions data. Objects in OSS are organized, partitioned, and distributed based on time. - When the path does not use time variables, all files are delivered to an OSS folder whose name contains a specific prefix.
Synchronization Mode	The mode in which to deliver data. The following options are available: - Incremental: Only incremental data is synchronized. - Full: All data in tables is scanned and synchronized. - Differential: Full data is synchronized before incremental data is synchronized. When you synchronize data in incremental mode, you can view the time when data is last delivered and the status of the current delivery task.
Destination Object Format	The delivered data is stored in the Parquet format. By default, data delivery uses PLAIN for encoding. PLAIN can be used to encode data of any type.

Parameter	Description
Schema Generation Type	Specify the columns for delivery. The order in which fields are sorted in the Tablestore table can be different from the order of fields in the schema. Parquet data stored in OSS is distributed based on the order of fields in the schema. Select a schema generation type. ■ If **Schema Generation Type:** is set to **Manual**, you must configure the source fields, destination field names, and destination field types for delivery. ■ If **Schema Generation Type:** is set to **Auto Generate**, the system identifies and matches the fields for delivery. **Notice** The data types must be consistent between the source and destination fields. Otherwise, the fields are discarded as dirty data. For more information about field type mappings, see data format mapping. When you configure the schema, you can perform the following operations: ■ Click **+Add Field** to add fields for delivery. ■ Click the e⌄ or ⌃ icon in the **Actions** column corresponding to a field to adjust the order of the field. ■ Click the 🗑 icon in the **Actions** column corresponding to a field to delete the field.
Schema Configurations	

6. Click **OK**.

   In the **View Statement to Create Table** message, you can view the statement that is used to create an external table for Data Lake Analytics (DLA) and E-MapReduce (EMR). You can also copy the statement to create an external table for DLA and EMR to access data in OSS.

   After the delivery task is created, you can perform the following operations:

   - View the details of the delivery task, such as the task name, table name, destination bucket, destination prefix, status, and the time when data is last synchronized.
   - View or copy the statement to create a table.

     Click **View Statement to Create Table** in the **Actions** column. You can view or copy the statement to create an external table by using computing engines such as EMR. For more information about specific operations, see Use EMR.
   - View the error message returned after the delivery.

If the configurations for the OSS bucket and delivery permissions are incorrect, data delivery cannot be complete. On the status page of the delivery task, you can view related error messages. For more information about exception handling, see Exception handling.

- Delete the delivery task.

  Click **Delete** in the **Actions** column corresponding to the delivery task. You can delete the delivery task. The system returns an error if the delivery task is in the initialization process. You can delete the task later.

View OSS data

After the delivery task is initialized and data is delivered, you can view the data delivered to OSS by using the OSS console, API, SDK, or computing engine EMR. For more information, see Overview.

Example of an OSS object URL:

```
oss://BucketName/TaskPrefix/TaskName_ConcurrentID_TaskPrefix__SequenceID
```

In the example, BucketName indicates the name of the bucket. The first TaskPrefix indicates the prefix of the destination folder. The second TaskPrefix indicates the prefix information of the task. TaskName indicates the name of the task. ConcurrentID indicates the number for concurrency determined by the system. The number starts from 0 and increases when the throughput increases. SequenceID indicates the sequence ID of the delivered file and increases from 1.

Partition data by time

Data delivery allows the system to obtain the time when data is written to Tablestore. The time consists of the following variables: $yyyy (four-digit year), $MM (two-digit month), $dd (two-digit day), $HH (two-digit hour), and $mm (two-digit minute). The time can be used as the prefix of the destination bucket after conversion.

Note

We recommend that the size of an OSS object be at least 4 MB. When computing engines are used to load OSS data, the larger number of partitions results in longer time to load partitions. Therefore, in most real-time data writing scenarios, we recommend that the time based on which to partition data is at the granularity of day or hour, rather than minute.

The delivery task by which data was written to Tablestore at 16:03 on August 31, 2020 is used in the example. The following table describes the URLs of the first object generated in OSS based on different destination prefix configurations.

OSS Bucket	TaskName	Destination prefix	OSS object URL
myBucket	testTask	myPrefix	oss://myBucket/myPrefix/testTask_0_myPrefix__1
myBucket	testTaskTimeParitioned	myPrefix/$yyyy/$MM/$dd/$HH/$mm	oss://myBucket/myPrefix/2020/08/31/16/03/testTaskTimeParitioned_0_myPrefix_2020_08_31_16_03__1

OSS Bucket	TaskName	Destination prefix	OSS object URL
myBucket	testTaskTimeParitionedHiveNamingStyle	myPrefix/year=$yyyy/month=$MM/day=$dd	oss://myBucket/myPrefix/year=2020/month=08/day=31/testTaskTimeParitionedHiveNamingStyle_0_myPrefix_year=2020_month=08
myBucket	testTaskDs	ds=$yyyy$MM$dd	oss://myBucket/ds=20200831/testTaskDs_0_ds=20200831__0

Data type mappings

Parquet Logical Type	Data type in Tablestore
Boolean	Boolean
Int64	Int64
Double	Double
UTF8	String

Exception handling

Error code	Cause	Solution
UnAuthorized	Tablestore is not authorized to deliver data to OSS.	Confirm whether the AliyunServiceRoleForOTSDataDelivery role exists. If the role does not exist, you must create a delivery task to trigger Tablestore to create the role.
InvalidOssBucket	The destination OSS bucket does not exist.	• Confirm whether the OSS bucket and Tablestore instance are located within the same region. • Confirm whether the OSS bucket exists. When the OSS bucket is created, all data is written to the OSS bucket again. The delivery progress is updated.

10.3. Use SDKs

Before you use SDKs to deliver data, you must understand the precautions and API operations. You can create a delivery task in the Tablestore console to deliver data from a Tablestore table to an OSS bucket.

Usage notes

- Data delivery is available in the China (Hangzhou), China (Shanghai), China (Beijing), and China (Zhangjiakou) regions.
- The delete operation on Tablestore data is ignored when the data is delivered. Tablestore data delivered to OSS is not deleted when you perform a delete operation on the data.
- It takes at most one minute for initialization when you create a delivery task.
- Latencies are within 3 minutes when data is written at a steady rate. P99 is within 10 minutes when data is synchronized.

  Note

  P99 indicates the average latency of the slowest 1% of requests over the last 10 seconds.

Operations

Operation	Description
CreateDeliveryTask	Creates a delivery task.
ListDeliveryTask	Lists all delivery task information of a table.
DescribeDeliveryTask	Queries the descriptive information of a delivery task.
DeleteDeliveryTask	Deletes a delivery task.

Use Tablestore SDKs

You can use the following Tablestore SDKs to implement data delivery:

- Java SDK
- Go SDK

Parameters

Parameter	Description
tableName	The name of the table.

Parameter	Description
taskName	The name of the delivery task. The name must be 3 to 16 characters in length and can contain only lowercase letters, digits, and hyphens (-). It must start and end with a lowercase letter or digit.

Parameter	Description
taskConfig	The configurations of the delivery task. Valid values: - ossPrefix: the prefix of the folder in the bucket. Data is delivered from Tablestore to the folder. The path of the destination folder supports the following time variables: $yyyy, $MM, $dd, $HH, and $mm. ○ When the path uses time variables for delivery, OSS folders are dynamically generated based on the time when data is written. This way, data is partitioned based on the naming conventions that are followed when Hive partitions data. Objects in OSS are organized, partitioned, and distributed based on time. ○ When the path does not use time variables, all files are delivered to an OSS folder whose name contains a specific prefix. - ossBucket: the name of the OSS bucket. - ossEndpoint: the endpoint of the region where an OSS bucket is located. - ossStsRole: the Alibaba Cloud Resource Name (ARN) of the Tablestore service-linked role. - format: the format of the delivered data. The delivered data is stored in the Parquet format. By default, data delivery uses PLAIN to encode data of any type. - eventTimeColumn: the event time column. This parameter specifies that data is partitioned based on the time of data in a column. If you do not specify this parameter, data is partitioned based on the time when the data is written to Tablestore. - parquetSchema: specifies the column you want to deliver. You must configure the source fields, destination fields, and destination field types to deliver. The order in which fields are sorted in Tablestore can be different from the order of fields in the schema. Parquet data stored in OSS is distributed based on the order of fields in the schema. Notice The data types must be consistent between the source and destination fields. If the data types between the fields are inconsistent, the fields are discarded as dirty data. For more information about field type mappings, see the "Data type mappings" section of the Quick start topic.

Parameter	Description
taskType	The mode in which to deliver data. Default value: BASE_INC. Valid values: ● INC: the incremental data delivery mode. Only incremental data is synchronized. ● BASE: the full data delivery mode. All data in tables is scanned and synchronized. ● BASE_INC: the differential data delivery mode. After the full data is synchronized, Tablestore synchronizes the incremental data. When you synchronize data in incremental mode, you can view the time when data is last delivered and the status of the current delivery task.

Examples

```
import com.alicloud.openservices.tablestore.ClientException;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.TableStoreException;
import com.alicloud.openservices.tablestore.model.delivery.*;
public class DeliveryTask {

        public static void main(String[] args) {
            final String endPoint = "https://yourinstancename.cn-hangzhou.ots.aliyuncs.com"
;

            final String accessKeyId = "LT********************g5";

            final String accessKeySecret = "Er**************************Yc";

            final String instanceName = "yourinstancename";

            SyncClient client = new SyncClient(endPoint, accessKeyId, accessKeySecret, inst
anceName);
            try {
                createDeliveryTask(client);
                System.out.println("end");
            } catch (TableStoreException e) {
                System.err.println("The operation failed. Details:" + e.getMessage() + e.ge
tErrorCode() + e.toString());
                System.err.println("Request ID:" + e.getRequestId());
            } catch (ClientException e) {
                System.err.println("The request failed. Details:" + e.getMessage());
            } finally {
                client.shutdown();
            }
        }

        private static void createDeliveryTask(SyncClient client){
            String tableName = "sampleTable";
            String taskName = "sampledeliverytask";
            OSSTaskConfig taskConfig = new OSSTaskConfig();
            taskConfig.setOssPrefix("sampledeliverytask/year=$yyyy/month=$MM");
            taskConfig.setOssBucket("datadeliverytest");
            taskConfig.setOssEndpoint("oss-cn-hangzhou.aliyuncs.com");
            taskConfig.setOssStsRole("acs:ram::17************45:role/aliyunserviceroleforot
sdatadelivery");
            // eventColumn is optional. This parameter specifies that data is partitioned b
ased on the time of data in a column. If you do not specify this parameter, data is partiti
oned based on the time when the data is written to Tablestore.
            EventColumn eventColumn = new EventColumn("Col1", EventTimeFormat.RFC1123);
            taskConfig.setEventTimeColumn(eventColumn);
            taskConfig.addParquetSchema(new ParquetSchema("PK1", "PK1", DataType.UTF8));
            taskConfig.addParquetSchema(new ParquetSchema("PK2", "PK2", DataType.BOOL));
            taskConfig.addParquetSchema(new ParquetSchema("Col1", "Col1", DataType.UTF8));
            CreateDeliveryTaskRequest request = new CreateDeliveryTaskRequest();
            request.setTableName(tableName);
            request.setTaskName(taskName);
            request.setTaskConfig(taskConfig);
            request.setTaskType(DeliveryTaskType.BASE_INC);
            CreateDeliveryTaskResponse response = client.createDeliveryTask(request);
            System.out.println("resquestID: "+ response.getRequestId());
            System.out.println("traceID: " + response.getTraceId());
            System.out.println("create delivery task success");
        }
}
```

10.4. Data lake-based computing and analysis

10.4.1. Use EMR

You can use E-MapReduce (EMR) JindoFS in cache mode to connect to Object Storage Service (OSS) that is used as the data lake.

Background information

You can use EMR JindoFS in cache or block storage mode to connect to OSS.

- When you use JindoFS in cache mode, files are stored as objects in OSS, and the frequently accessed objects are cached on the local disk of an EMR cluster to improve the data access efficiency. In cache mode, JindoFS can access objects in OSS without the need to convert the object formats. JindoFS is fully compatible with OSS clients. For more information, see Use JindoFS in cache mode.
- The block storage mode ensures efficient read and write operations and high metadata accessibility. JindoFS uses OSS as the storage backend. In block storage mode, JindoFS stores data as blocks in OSS and uses Namespace Service to maintain metadata. This ensures high performance when you read and write data or query metadata. For more information, see Use JindoFS in block storage mode.

Prerequisites

- An EMR cluster is created. For more information, see Create a cluster.

  Before you create a cluster, take note of the following items:

- The EMR cluster and OSS belong to the same Alibaba Cloud account. We recommend that the EMR cluster and OSS bucket are located in the same region.
- When you create a cluster, turn on **Assign Public IP Address** to access the cluster over the Internet and **Remote Logon** to log on to a remote server by using Shell.
- SmartData and Bigboot are dependent services to use configurations of JindoFS. If these services are not selected by default, select these services.

- A delivery task is created. For more information, see Quick start.

Procedure

1. Use EMR JindoFS in cache mode to connect to OSS and enable local cache. For more information, see Use JindoFS in cache mode.

   After you enable local cache, hot data blocks are cached on local disks. By default, this feature is disabled, which indicates that EMR directly reads data from OSS. After you enable local cache, Jindo automatically manages cached data. Jindo clears cached data based on the disk space usage you configured. Configure the usage to clear cached data and adjust the space usage of local disks.

2. Start Spark SQL.

   i. Use remote logon tools such as PuTTY to log on to the EMR Header server.

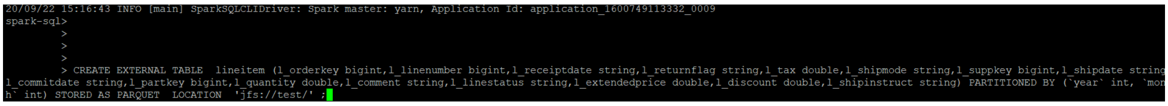
   ii. Run the following command to run Spark SQL:

   ```
   spark-sql --master yarn --num-executors 5 --executor-memory 1g --executor-cores 2
   ```

3. Use an SQL statement to create a external table that maps to an OSS folder.

   Obtain an SQL statement in the Tablestore console. The following example of an SQL statement is used only for reference:

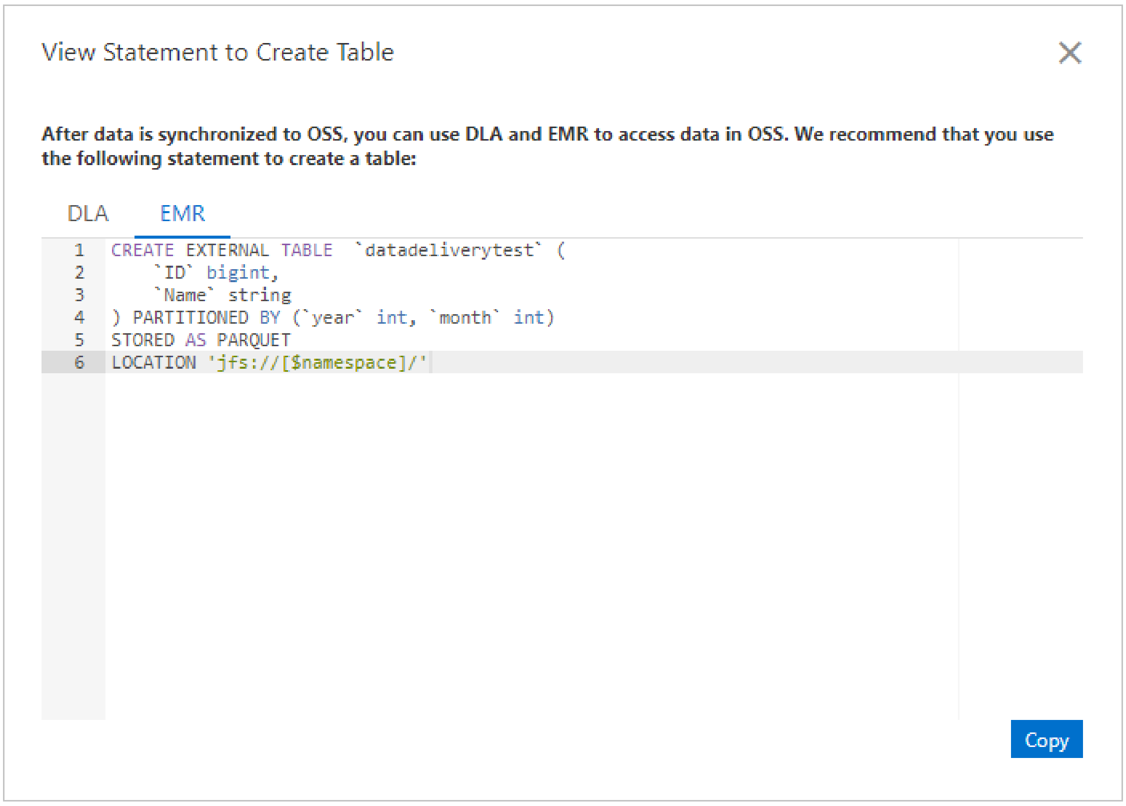
   ```
   CREATE EXTERNAL TABLE  lineitem (l_orderkey bigint,l_linenumber bigint,l_receiptdate string,l_returnflag string,l_tax double,l_shipmode string,l_suppkey bigint,l_shipdate string,l_commitdate string,l_partkey bigint,l_quantity double,l_comment string,l_linestatus string,l_extendedprice double,l_discount double,l_shipinstruct string) PARTITIONED BY (`year` int, `month` int) STORED AS PARQUET  LOCATION  'jfs://test/' ;
   ```



```
20/09/22 15:16:43 INFO [main] SparkSQLCLIDriver: Spark master: yarn, Application Id: application_1600749113332_0009
spark-sql>
         >
         >
         >
         > CREATE EXTERNAL TABLE  lineitem (l_orderkey bigint,l_linenumber bigint,l_receiptdate string,l_returnflag string,l_tax double,l_shipmode string,l_suppkey bigint,l_shipdate string
l_commitdate string,l_partkey bigint,l_quantity double,l_comment string,l_linestatus string,l_extendedprice double,l_discount double,l_shipinstruct string) PARTITIONED BY (`year` int, `mon
h` int) STORED AS PARQUET  LOCATION  'jfs://test/' ;
```

   To obtain an SQL statement in the Tablestore console, use the following method:

   On the **Deliver Data to OSS** tab, click **View SQL Statement to Create Table** in the **Actions** column corresponding to a delivery task. You can view and copy the SQL statement. The following figure shows an example of an SQL statement that is used to create a external table.



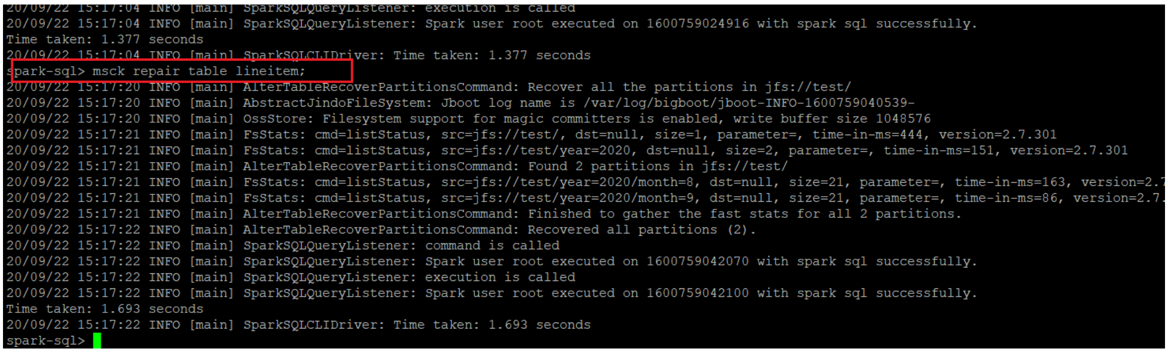
After data is synchronized to OSS, you can use DLA and EMR to access data in OSS. We recommend that you use the following statement to create a table:

```
CREATE EXTERNAL TABLE `datadeliverytest` (
    `ID` bigint,
    `Name` string
) PARTITIONED BY (`year` int, `month` int)
STORED AS PARQUET
LOCATION 'jfs://[$namespace]/'
```

4. Execute the following SQL statement to load data partitions from an OSS source.

   lineitem is the name of the external table.

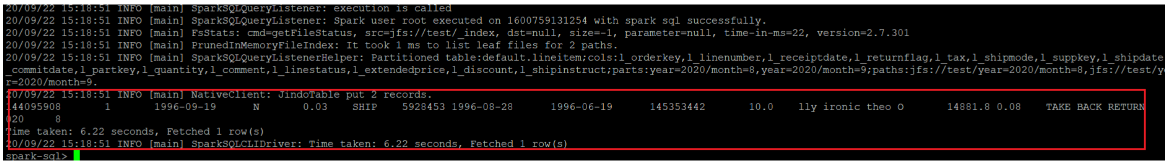
```
msck repair table lineitem;
```



```
20/09/22 15:17:04 INFO [main] SparkSQLQueryListener: execution is called
20/09/22 15:17:04 INFO [main] SparkSQLQueryListener: Spark user root executed on 1600759024916 with spark sql successfully.
Time taken: 1.377 seconds
20/09/22 15:17:04 INFO [main] SparkSQLCLIDriver: Time taken: 1.377 seconds
spark-sql> msck repair table lineitem;
20/09/22 15:17:20 INFO [main] AlterTableRecoverPartitionsCommand: Recover all the partitions in jfs://test/
20/09/22 15:17:20 INFO [main] AbstractJindoFileSystem: Jboot log name is /var/log/bigboot/jboot-INFO-1600759040539-
20/09/22 15:17:20 INFO [main] OssStore: Filesystem support for magic committers is enabled, write buffer size 1048576
20/09/22 15:17:21 INFO [main] FsStats: cmd=listStatus, src=jfs://test/, dst=null, size=1, parameter=, time-in-ms=444, version=2.7.301
20/09/22 15:17:21 INFO [main] FsStats: cmd=listStatus, src=jfs://test/year=2020, dst=null, size=2, parameter=, time-in-ms=151, version=2.7.301
20/09/22 15:17:21 INFO [main] AlterTableRecoverPartitionsCommand: Found 2 partitions in jfs://test/
20/09/22 15:17:21 INFO [main] FsStats: cmd=listStatus, src=jfs://test/year=2020/month=8, dst=null, size=21, parameter=, time-in-ms=163, version=2.7
20/09/22 15:17:21 INFO [main] FsStats: cmd=listStatus, src=jfs://test/year=2020/month=9, dst=null, size=21, parameter=, time-in-ms=86, version=2.7.
20/09/22 15:17:21 INFO [main] AlterTableRecoverPartitionsCommand: Finished to gather the fast stats for all 2 partitions.
20/09/22 15:17:22 INFO [main] AlterTableRecoverPartitionsCommand: Recovered all partitions (2).
20/09/22 15:17:22 INFO [main] SparkSQLQueryListener: command is called
20/09/22 15:17:22 INFO [main] SparkSQLQueryListener: Spark user root executed on 1600759042070 with spark sql successfully.
20/09/22 15:17:22 INFO [main] SparkSQLQueryListener: execution is called
20/09/22 15:17:22 INFO [main] SparkSQLQueryListener: Spark user root executed on 1600759042100 with spark sql successfully.
Time taken: 1.693 seconds
20/09/22 15:17:22 INFO [main] SparkSQLCLIDriver: Time taken: 1.693 seconds
spark-sql>
```

5. Query data.

```
select * from lineitem limit 1;
```



```
20/09/22 15:18:51 INFO [main] SparkSQLQueryListener: execution is called
20/09/22 15:18:51 INFO [main] SparkSQLQueryListener: Spark user root executed on 1600759131254 with spark sql successfully.
20/09/22 15:18:51 INFO [main] FsStats: cmd=getFileStatus, src=jfs://test/_index, dst=null, size=-1, parameter=null, time-in-ms=22, version=2.7.301
20/09/22 15:18:51 INFO [main] PrunedInMemoryFileIndex: It took 1 ms to list leaf files for 2 paths.
20/09/22 15:18:51 INFO [main] SparkSQLQueryListenerHelper: Partitioned table:default.lineitem;cols:l_orderkey,l_linenumber,l_receiptdate,l_returnflag,l_tax,l_shipmode,l_suppkey,l_shipdate
_commitdate,l_partkey,l_quantity,l_comment,l_linestatus,l_extendedprice,l_discount,l_shipinstruct;parts:year=2020/month=8,year=2020/month=9;paths:jfs://test/year=2020/month=8,jfs://test/y
r=2020/month=9.
20/09/22 15:18:51 INFO [main] NativeClient: JindoTable put 2 records.
144095908      1       1996-09-19      N       0.03    SHIP    5928453 1996-08-28      1996-06-19      145353442       10.0    lly ironic theo O       14881.8 0.08    TAKE BACK RETURN
020     8
Time taken: 6.22 seconds, Fetched 1 row(s)
20/09/22 15:18:51 INFO [main] SparkSQLCLIDriver: Time taken: 6.22 seconds, Fetched 1 row(s)
spark-sql>
```

11.Data visualization

11.1. Data visualization tools

This topic describes data visualization tools to which you can connect Tablestore.

Tool	Description	References
DataV	DataV can convert statistic data to a variety of dynamic visualization charts. For more information, see What is DataV?. You can use DataV to display data in data tables or secondary indexes of Tablestore. In most cases, DataV is used to build enterprise application systems for complex big data processing and analytics.	Connect Tablestore to DataV
Grafana	Grafana is an open source visualization and analytics platform that supports data query and visualization for various data sources such as Prometheus, Graphite, OpenTSDB, InfluxDB, Elasticsearch, MySQL, and PostgreSQL. For more information, see the Grafana official documentation. You can use Grafana to display data in data tables or time series tables of Tablestore.	Connect Tablestore to Grafana

11.2. Connect Tablestore to Grafana

After Tablestore is connected to Grafana, you can use Grafana to display Tablestore data.

Prerequisites

- The first time you use Tablestore, you must first activate Tablestore and create instances and data tables. For more information, see Manage the Wide Column model in the Tablestore console or Manage the Wide Column model in the Tablestore CLI.
- A mapping table is created for a data table or a time series table in Tablestore. For more information about specific operations, see Create mapping tables for tables and Create mapping tables in the multi-value model for time series tables.
- Open source Grafana whose version is later than 8.0.0 is installed. For more information about how to install Grafana, see the Grafana official documentation.
- An AccessKey pair that consists of an AccessKey ID and an AccessKey secret is obtained. For more information, see Obtain an AccessKey pair.

Background information

Grafana is an open source visualization and analytics platform that supports data query and visualization for various data sources such as Prometheus, Graphite, OpenTSDB, InfluxDB, Elasticsearch, MySQL, and PostgreSQL. For more information, see the Grafana official documentation.

After you add a Tablestore data source to Grafana, Grafana displays real-time data on a dashboard based on the data of Tablestore.

Usage notes

The support for Grafana is available in the following regions: China (Hangzhou), China (Shanghai), China (Beijing), China (Zhangjiakou), China (Shenzhen), and Singapore (Singapore).

Step 1: Install the Grafana plug-in for Tablestore

Windows

1. Click Grafana plug-in for Tablestore package to download the Grafana plug-in for Tablestore package.
2. Extract files from the package to the *plugins-bundled* directory of the Grafana plug-in.
3. Modify the configuration file of Grafana.

   i. Use a text editor to open the defaults.ini configuration file in the conf directory of the Grafana plug-in.

   ii. Find **[plugins]** in the configuration file and configure the allow_loading_unsigned_plugins parameter.

   ```
   allow_loading_unsigned_plugins = aliyun-tablestore-grafana-datasource
   ```

4. Restart the grafana-server.exe process in Task Manager.

Linux or macOS

1. Run the following command to download the Grafana plug-in for Tablestore package:

   ```
   wget https://tablestore-doc.oss-cn-hangzhou.aliyuncs.com/aliyun-tablestore-grafana-plug
   in/tablestore-grafana-plugin-1.0.0.zip
   ```

2. Extract files from the Grafana plug-in for Tablestore package to a directory of the Grafana plug-in.

   Run the following command to extract files from the Grafana plug-in for Tablestore package based on the method that is used to install the Grafana plug-in.

   - If you use a yum repository or an RPM package to install the Grafana plug-in in Linux, run the **unzip tablestore-grafana-plugin-1.0.0.zip -d /var/lib/grafana/plugins** command.
   - If you use a .tar.gz file to install the Grafana plug-in, run the **unzip tablestore-grafana-plugin-1.0.0.zip -d {PATH_TO}/grafana-{VERSION}/data/plugins** command.

3. Modify the configuration file of Grafana.

   i. Go to the directory of Grafana and open the configuration file.

   - If you use a yum repository or an RPM package to install Grafana in Linux, open the */etc/grafana/grafana.ini* file.
   - If you use a .tar.gz file to install Grafana, open the *{PATH_TO}/grafana-{VERSION}/conf/defaults.ini* file.

     In the preceding file name, `{PATH_TO}/grafana-{VERSION}` indicates the installation path of Grafana and VERSION indicates the version number of Grafana.

   ii. Find **[plugins]** in the configuration file and configure the allow_loading_unsigned_plugins parameter.

   ```
   allow_loading_unsigned_plugins = aliyun-tablestore-grafana-datasource
   ```

4. Restart Grafana.
    1. Run the **kill** command to terminate the Grafana process.
    2. Run the following command to start Grafana.
        - If you use a yum repository or an RPM package to install Grafana, run the **systemctl restart grafana-server** command.
        - If you use a .tar.gz file to install Grafana, run the **./bin/grafana-server web** command.

Step 2: Add a data source

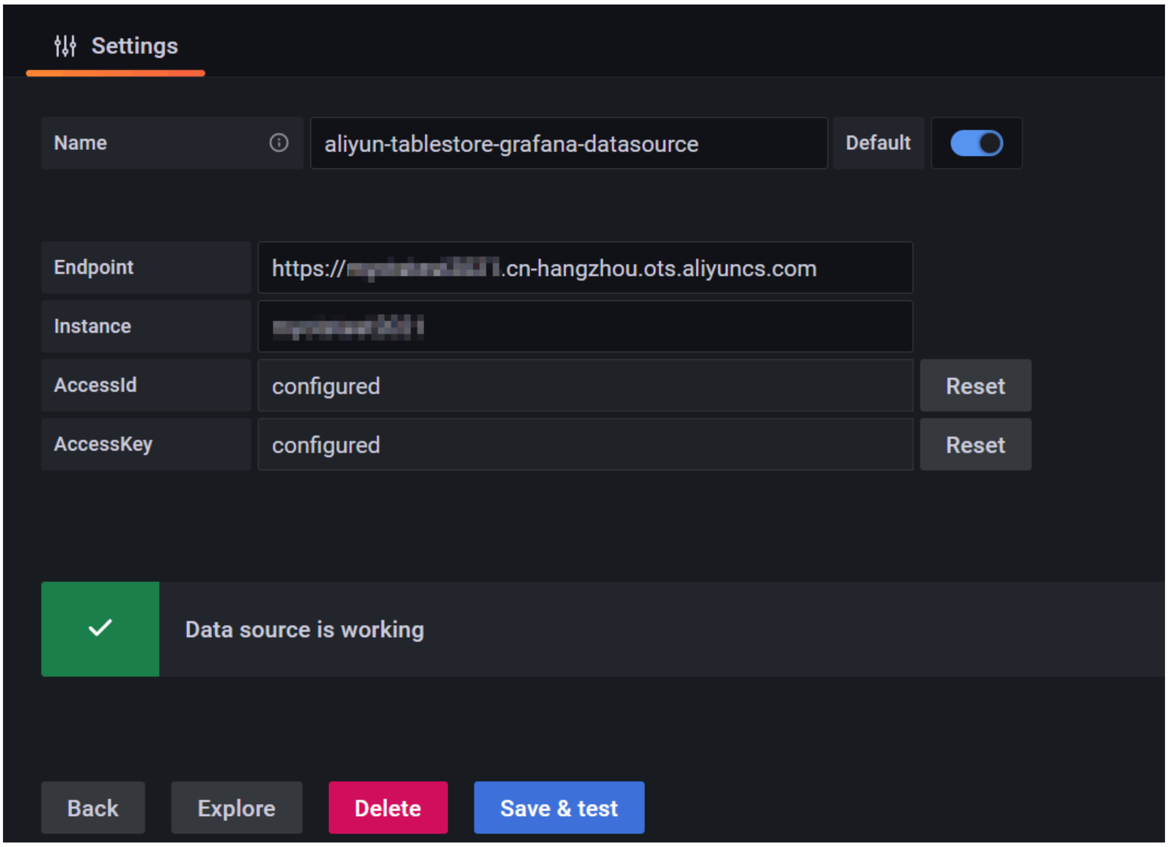
1. Log on to Grafana.
    1. Open a browser and enter `http://localhost:3000/` in the address box to go to the Grafana logon page.
    2. Enter values for the Email or username and Password parameters, and click **Log in**.

       The default username and password that are used to log on to Grafana are both admin. The first time you log on to Grafana, change the default password as prompted.
2. Move the pointer over the ⚙ icon in the left-side navigation pane and click **Data sources**.
3. On the **Data sources** tab, click **Add data source**.
4. In the Others section of the **Add data source** page, click aliyun-tablestore-grafana-datasource.
5. On the **Settings** page, configure the parameters. The following table describes the parameters.

Parameter	Example	Description
Name	aliyun-tablestore-grafana-datasource	The name of the data source. You can retain the default value or enter a custom name. Default value: aliyun-tablestore-grafana-datasource.
Endpoint	https://myinstance.cn-hangzhou.ots.aliyuncs.com	The endpoint of the Tablestore instance. Configure this parameter based on the Tablestore instance that you want to access. For more information, see Endpoint.
Instance	myinstance	The name of the Tablestore instance.
AccessId	************************	The AccessKey ID of your Alibaba Cloud account or a RAM user that has permissions to access Tablestore.
AccessKey	********************************	The AccessKey secret of your Alibaba Cloud account or a RAM user that has permissions to access Tablestore.

6. Click **Save & test**.

   After the data source is added to Grafana, the **Data source is working** message appears.



Settings

Name: aliyun-tablestore-grafana-datasource

Default: ●

Endpoint: https://[redacted].cn-hangzhou.ots.aliyuncs.com

Instance: [redacted]

AccessId: configured

AccessKey: configured

✓ Data source is working

Step 3: Create a dashboard panel

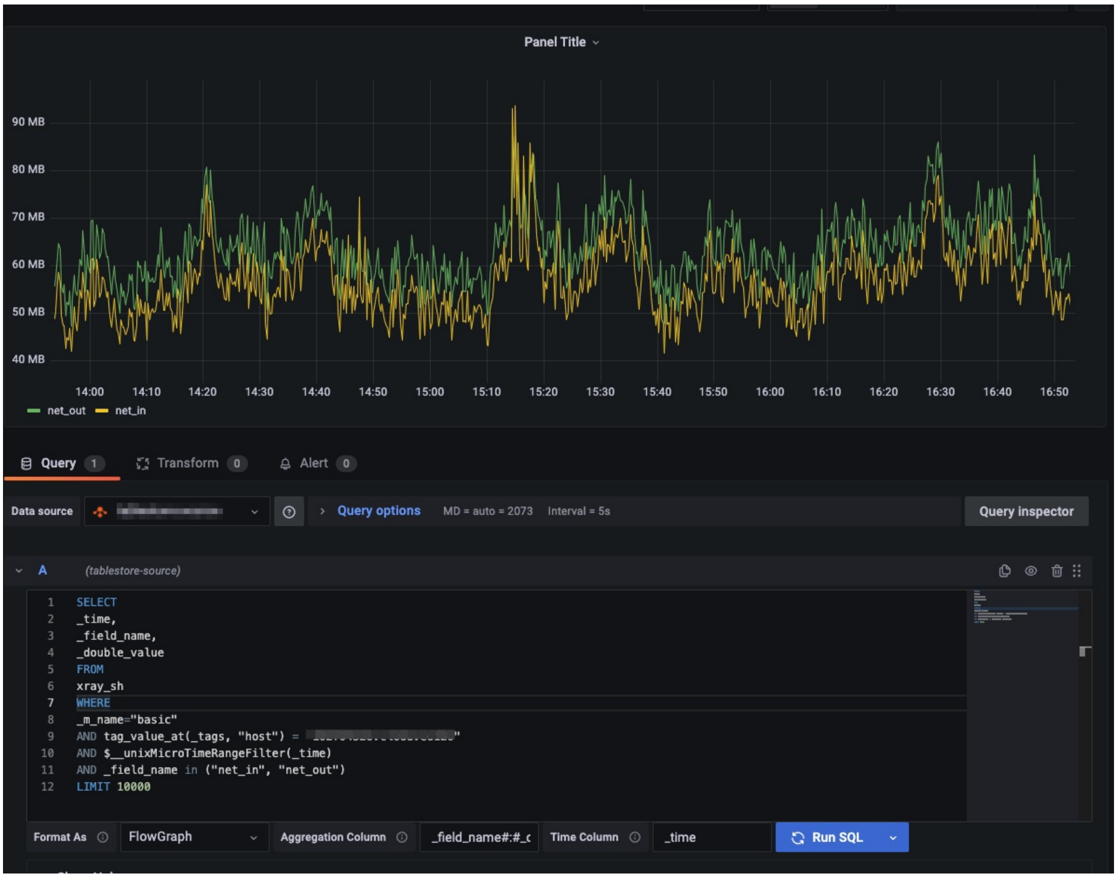
1. In the Grafana console, move the pointer over the + icon in the left-side navigation pane and click **Dashboard**.
2. On the **New dashboard** page, click the icon.
3. In the **Add panel** section, click **Add a new panel**.
4. In the **Query** section of the **Edit Panel** page, specify the conditions that are used to query data from the data source.
   i. Select a Tablestore data source from the Data source drop-down list.
   ii. Configure the data source parameters.

Parameter	Example	Description

Parameter	Example	Description
Query	`SELECT * FROM your_table WHERE $__unixMicroTimeRangeFilter(_time) AND _m_name = "your_measurement" AND tag_value_at(_tags, "your_tag")="your_tag_value"LIMIT 1000`	The SQL query statement. For more information, see Query data. Notice - In the WHERE clause, the time range condition that is used to filter data is specified by using a predefined macro. In this example, the predefined macro is `$__unixMicroTimeRangeFilter`. You can click Show Help on the configuration page to view more time macro functions. - If you want data to be displayed as a time series chart, you must specify that the time column that is represented by a numeric timestamp must be returned and specify the name of the time column.
Format As	Timeseries	The method that you want to use to display the query results. Default value: Timeseries. Valid values: - Timeseries: displays the query results as a regular time series chart. - FlowGraph: displays the query results as a multi-dimensional chart. - Table: displays the query results as a regular table.
Time Column	_time	The name of the time column in the query results that are returned. The time column is used as the x-axis of the time series chart. If you set the Format As parameter to **Timeseries** or **FlowGraph**, you can configure this parameter.

Parameter	Example	Description
Aggregation Column	_field_name#:#_double_value	The name of the single column whose data in multiple rows at a specified point in time you want to convert into data in multiple columns of a single row at the specified point in time. This parameter is used if you want to convert data in single-value model to data in multi-value model. The data in single-value model is the results that are obtained when you use SQL to query a time series table in Tablestore. If you set the Format As parameter to **FlowGraph**, you can configure this parameter. The value of this parameter is in the `<Data point name column>#:#<Data value column>` format.

5. Click **Run SQL** to execute SQL statements, view the data, and then perform debugging.
6. Configure the dashboard panel and save the configurations.
   i. On the right side of the page, specify the name, type, and layout of the monitoring chart.



Data source [redacted]

```
SELECT
_time,
_field_name,
_double_value
FROM
xray_sh
WHERE
_m_name="basic"
AND tag_value_at(_tags, "host") = [redacted]"
AND $__unixMicroTimeRangeFilter(_time)
AND _field_name in ("net_in", "net_out")
LIMIT 10000
```

Format As: FlowGraph
Aggregation Column: _field_name#:#_c
Time Column: _time

   ii. In the upper-right corner, click **Apply**.
   iii. Click the 🖫 icon in the upper-right corner. In the **Save dashboard as...** dialog box, configure the Dashboard name and Folder parameters and click **Save**.

Step 4: View monitoring data

1. In the Grafana console, move the pointer over the ⊞ icon in the left-side navigation pane and click **Browse**.
2. On the **Browse** tab, click the monitoring dashboard under the folder that you specified for the dashboard to view all monitoring charts on the dashboard.

11.3. Connect Tablestore to DataV

After you add a Tablestore data source in the DataV console, you can use DataV to display Tablestore data.

Prerequisites

- The first time you use Tablestore, you must first activate Tablestore and create instances and data tables. For more information, see Manage the Wide Column model in the Tablestore console or Manage the Wide Column model in the Tablestore CLI.
- The first time you use DataV, you must first activate DataV. For more information, see Activate DataV.
- An AccessKey pair that consists of an AccessKey ID and an AccessKey secret is obtained. For more information, see Obtain an AccessKey pair.

Context

DataV can convert statistic data to a variety of dynamic visualization charts. For more information, see What is DataV?.

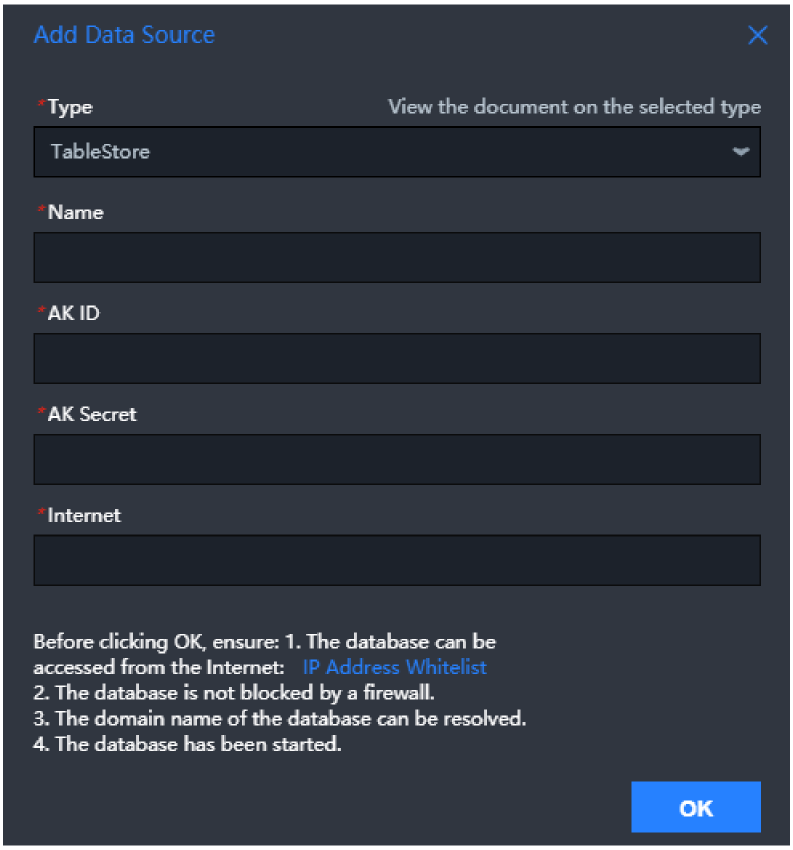
After you add a Tablestore data source to DataV, DataV displays real-time data on a dashboard based on the data of Tablestore tables.

Usage notes

The data source can only be a data table or a secondary index in Tablestore.

Step 1: Add a Tablestore data source

1. Log on to the DataV console.
2. On the **Data Sources** tab, click **Data Sources**.
3. On the **Data Sources** page, click **Add Source**.



*Type

TableStore

*Name

*AK ID

*AK Secret

*Internet

Before clicking OK, ensure: 1. The database can be accessed from the Internet: IP Address Whitelist
2. The database is not blocked by a firewall.
3. The domain name of the database can be resolved.
4. The database has been started.

4. In the **Add Data Source** dialog box, select **TableStore** from the Type drop-down list. Then, configure other parameters for a Tablestore data source. The following table describes the parameters that you can configure for a Tablestore data source.

Parameter	Description
Name	The display name of the data source.
AK ID	The AccessKey ID of your Alibaba Cloud account or a RAM user that has permissions to access Tablestore.
AK Secret	The AccessKey secret of your Alibaba Cloud account or a RAM user that has permissions to access Tablestore.
Internet	The endpoint of the Tablestore instance. Configure this parameter based on the Tablestore instance that you want to access. For more information, see Endpoint.

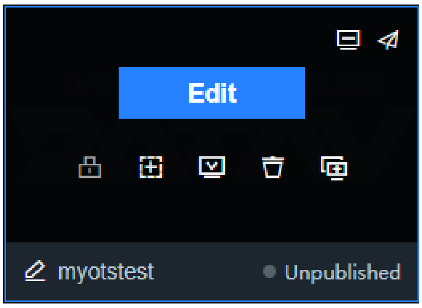
5. Click **OK**.

   The data source that you added is displayed in the data source list.

Step 2: Configure the Tablestore data source

1. Log on to the DataV console.
2. Create a visualization project or select an existing visualization project based on your business scenario.
   - If you use an existing visualization project, move the pointer over the visualization project and

click **Edit**.



- If this is the first time that you use DataV or existing visualization projects do not meet your requirements, you can create visualization projects based on your business requirements.

  > **Note** You can create visualization projects on a PC or mobile device, or by using image recognition. The following procedure describes how to create a visualization project on a PC.

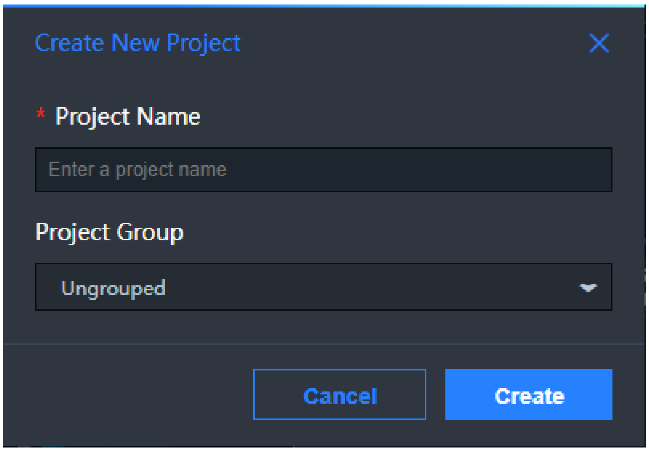
  a. On the **Projects** tab, click **PC Creation**.

  b. Move the pointer over the visualization application for which you want to create a project and click **Create Project**.

     DataV allows you to create a visualization application by using a template or a blank canvas. You can select a method to create a visualization application based on your business requirements.

  c. In the **Create New Project** dialog box, specify the name of the project and select a group to which the project belongs.

  d. Click **Create**.



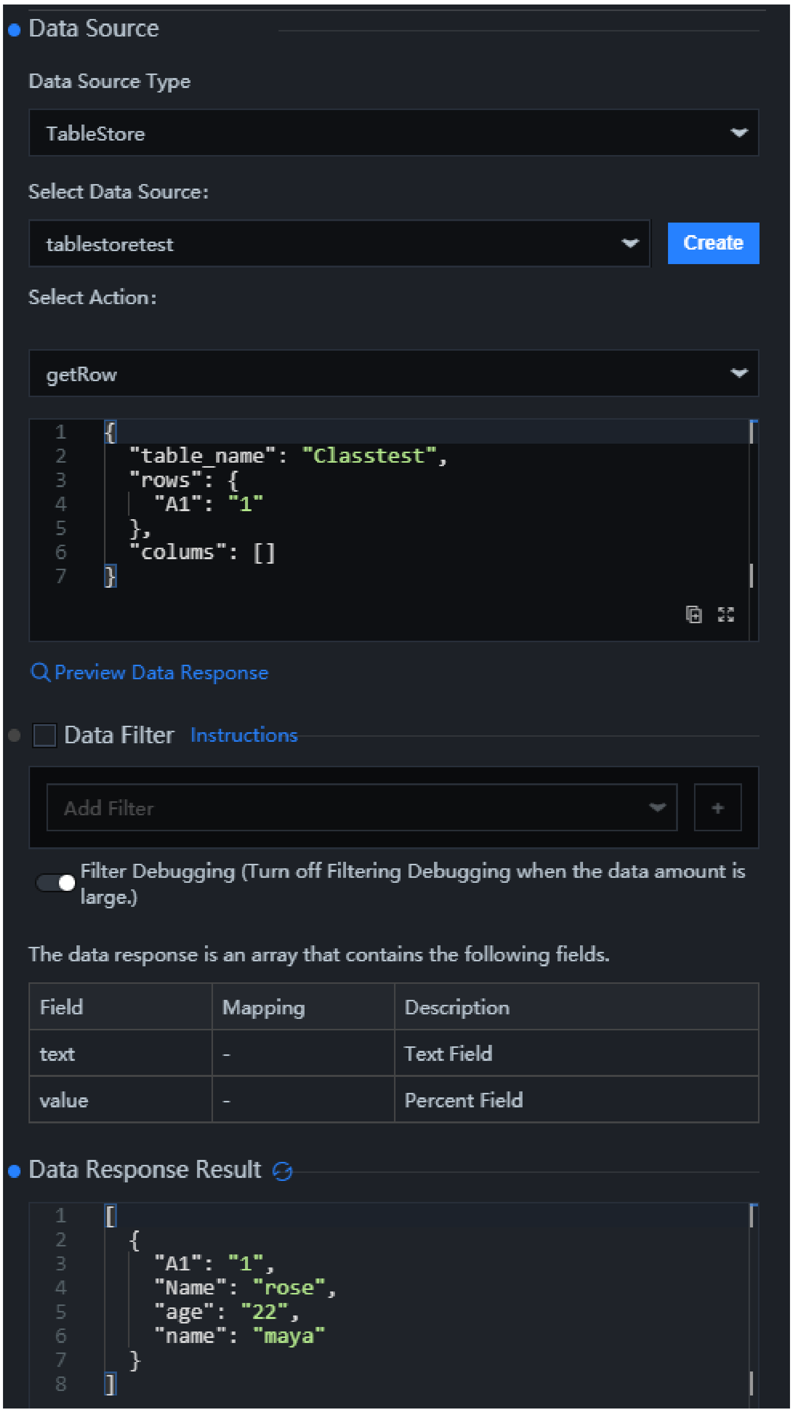
3. Configure the Tablestore data source.

   i. On the canvas editing page, click a widget on the canvas.

   > **Note** If no widget exists on the canvas, add a widget first. For more information, see Add a widget.

   ii. In the right-side widget configurations panel of the canvas, click the [–] icon.

iii. On the **Data** tab, click **Set**.



Data Source

Data Source Type

TableStore

Select Data Source:

tablestoretest

Select Action:

getRow

```
{
  "table_name": "Classtest",
  "rows": {
    "A1": "1"
  },
  "colums": []
}
```

□ Data Filter

Filter Debugging (Turn off Filtering Debugging when the data amount is large.)

The data response is an array that contains the following fields.

Field	Mapping	Description
text	-	Text Field
value	-	Percent Field

Data Response Result

```
[
  {
    "A1": "1",
    "Name": "rose",
    "age": "22",
    "name": "maya"
  }
]
```

iv. Set the Data Source Type parameter to **TableStore** and select an existing data source.

v. Select an operation that you want to perform on the data source and enter a query statement.

Tablestore supports getRow and getRange. getRow corresponds to the GetRow operation in Tablestore. getRange corresponds to the GetRange operation in Tablestore.

- If you select getRow, a row whose primary key is specified is read. The following code and table describe the format that you can use and the parameters that you can configure for a query statement.

```
{
"table_name": "test",
"rows": {
"id": 2
},
"columns": [
"id",
"test"
]
}
```

Parameter	Description
table_name	The name of the table in Tablestore.
rows	The primary key of the row. **Notice** You must specify the same number and data types of primary key columns for each row as the number and data types of primary key columns in the table.
columns	The columns that you want to read. You can specify the names of primary key columns or attribute columns. If you do not specify a column name, all data in the row is returned.

- If you select getRange, all data whose primary keys are within a specified range is read. The following code and table describe the format that you can use and the parameters that you can configure for a query statement.

```
{
"table_name": "test",
"direction": "FORWARD",
"columns": [
"id",
"test"
],
"range": {
"limit": 4,
"start": {
"id": "InfMin"
},
"end": {
"id": 3
}
}
}
```

Parameter	Description
table_name	The name of the table in Tablestore.
direction	The direction in which data is read. ■ If you set this parameter to FORWARD, specify a smaller value for the start primary key than the value of the end primary key. The returned rows are sorted in ascending order based on their primary key values. ■ If you set this parameter to BACKWARD, specify a greater value for the start primary key than the value of the end primary key. The returned rows are sorted in descending order based on their primary key values. For example, if you set the direction parameter to FORWARD for a table that contains two primary keys A and B and the value of A is smaller than the value of B, the rows whose primary key values are greater than or equal to the value of A but smaller than the value of B are returned in ascending order from A to B. If you set the direction parameter to BACKWARD, the rows whose primary key values are smaller than or equal to the value of B and greater than the value of A are returned in descending order from B to A.

Parameter	Description
columns	The columns that you want to read. You can specify the names of primary key columns or attribute columns. If you do not specify a column name, all data in the row is returned. If a row is within the specified range to be read based on the primary key value but does not contain the specified columns to return, the response excludes the row.
limit	The maximum number of rows that you want to return. The value of this parameter must be greater than 0. An operation stops after the maximum number of rows that you want to return in the forward or backward direction is reached, even if some rows within the specified range are not returned.
start	The start and end primary keys of the range to read. The start and end primary keys must be valid primary keys or virtual points that consist of the InfMin and InfMax type data. The number of columns for each virtual point must be the same as the number of columns of each primary key. InfMin indicates an infinitely small value. All values of other types are greater than the InfMin type value. InfMax indicates an infinitely great value. All values of other types are smaller than the InfMax type value. **Notice** You must specify the same number and data types of primary key columns for each row as the number and data types of primary key columns in the table. ▪ start indicates the start primary key. If the row that contains the start primary key exists, the row of data is returned. ▪ end indicates the end primary key. No matter whether the row that contains the end primary key exists, the row of data is not returned.
end	

vi. Click the ⟳ icon next to **Data Response Result** to obtain the response.

> **Note** After you obtain the response, you can click **Preview Data Response** to view the response.

4. Preview and publish a project

   i. In the upper-right corner of Canvas Editor, click the ⊟ icon to preview the project.

ii. In the upper-right corner of Canvas Editor, click the icon.

iii. In the **Publish** dialog box, click **Publish Project**.

iv. In the **Publish Success** message, click **Cancel**.

> **Note** After the project is published, the content on the publish page is locked. After you edit the content, you can synchronize the published content by using the snapshot management feature. You can also click **Goto Snapshot Management** to view information about snapshots that are created.

v. Click the icon on the right side of the URL in the **Project URL** section.

vi. Copy and paste the URL to the address bar of a browser to view the published project.

12.Backup and restoration

12.1. Overview

Hybrid Backup Recovery (HBR) can be used to regularly back up data in Tablestore instances and restore lost or damaged data. HBR supports the backup of full data and incremental data and data redundancy. This improves the reliability of data in backup vaults.

Public preview

The data backup and restoration feature in Tablestore by using HBR is in public preview in the China (Shanghai) and China (Beijing) regions.

If you have questions, please join the Tablestore technical support group by searching for group ID 11789671 or 23307953 in DingTalk to contact us.

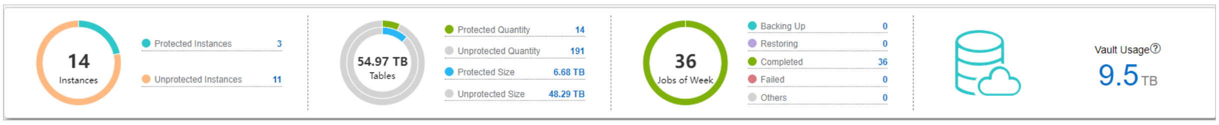
Background information

HBR is a unified platform that is developed by Alibaba Cloud for backup and disaster recovery. This platform is an easy-to-use data management service that is deployed in the public cloud to offer high agility, efficiency, security, and reliability. You can use HBR to back up data to a cloud vault from Elastic Computing Service (ECS) instances, ECS databases, file systems, NAS clusters, Object Storage Service (OSS) buckets, Tablestore, and self-managed data centers that store files, databases, virtual machines (VMs), and large-scale NAS file systems. You can also perform disaster recovery and archive data based on the archive policies that you configure for the preceding resources. For more information, see What is Hybrid Backup Recovery?.

Prerequisites

- HBR is activated.
- A Tablestore instance is created, and the data that you want to back up is available.

Backup overview



14 Instances

Protected Instances 3

Unprotected Instances 11

54.97 TB Tables

Protected Quantity 14

Unprotected Quantity 191

Protected Size 6.68 TB

Unprotected Size 48.29 TB

36 Jobs of Week

Backing Up 0

Restoring 0

Completed 36

Failed 0

Others 0

Vault Usage

9.5 TB

12.2. Back up Tablestore data

This topic describes how to use Hybrid Backup Recovery (HBR) to regularly back up full data or incremental data in Tablestore.

Usage notes

- The first time that you use this feature, HBR automatically creates the service-linked role AliyunServiceRoleForHbrOtsBackup to obtain Tablestore instances in your account. Follow the on-screen instructions to create the role. For more information, see Service-linked roles for HBR.
- By default, HBR reads the Tablestore instances in the region where HBR is deployed and automatically loads the instances without installing a client.
- You can back up data in Tablestore only to HBR backup vaults in the same region as the Tablestore

instance in which the data that you want to back up resides. Cross-region backup is not supported.

- You can back up and restore only data tables in Tablestore instances. You cannot back up indexes or time series tables.

12.3. Restore Tablestore data

If an exception occurs or an incorrect operation is performed on a Tablestore instance, you can restore data backups in the backup vaults to the Tablestore instance or to another Tablestore instance in the same region. This topic describes how to use Hybrid Backup Recovery (HBR) to create a Tablestore restoration task.

Prerequisites

A Tablestore backup plan is created and data backup is complete. For more information, see Back up Tablestore data.

13.Limits

13.1. General limits

This topic describes the limits of Tablestore. Table schemas and row sizes can be tailored to improve performance.

Limits on instances

Resource	Limit	Description
Number of instances created within an Alibaba Cloud account	10	If your business requirements are not met due to the limit, submit a ticket.
Number of tables in an instance	64	The number of data tables and index tables. If your business requirements are not met due to the limit, submit a ticket.
Length of an instance name	3 to 16 bytes	The name of an instance can contain letters, digits, and hyphens (-). The name must start with a letter and cannot end with a hyphen (-).

Limits on tables

Resource	Limit	Description
Length of a table name	1 to 255 bytes	The name of a table can contain letters, digits, and underscores (_). The name must start with a letter or an underscore (_).
Reserved read capacity units (CUs) and reserved write CUs of a single table	0 to 100000 CUs	If your business requirements are not met due to the limit, submit a ticket.
Number of predefined columns	0 to 32	Predefined columns are non-primary key columns whose names and types are defined when a data table is created. When you create a global secondary index, predefined columns can be used as the indexed columns or attribute columns of the index table. For more information, see Global secondary index. **Notice** Predefined columns are not required when you use a search index.

Limits on columns

Resource	Limit	Description
Length of a column name	1 to 255 bytes	The name of a table can contain letters, digits, and underscores (_). The name must start with a letter or an underscore (_).
Number of columns in a primary key	One to four	A primary key can contain one to four primary key columns.
Size of the value in a STRING primary key column	1 KB	The size of the value in a STRING primary key column cannot exceed 1 KB.
Size of the value in a STRING attribute column	2 MB	The size of the value in a STRING attribute column cannot exceed 2 MB.
Size of the value in a BINARY primary key column	1 KB	The size of the value in a BINARY primary key column cannot exceed 1 KB.
Size of the value in a BINARY attribute column	2 MB	The size of the value in a BINARY attribute column cannot exceed 2 MB.

Limits on rows

Resource	Limit	Description
Number of attribute columns in a single row	Tablestore does not impose limits on the number of attribute columns in a single row.	None.
Size of a single row	Tablestore does not impose limits on the size of a single row.	Tablestore does not impose limits on the total size of column names or column values for a row.

Limits on operations

Operation	Limit	Description
Number of attribute columns written by one request	1,024	During a PutRow, UpdateRow, or BatchWriteRow operation, the number of attribute columns written in a row cannot exceed 1,024.
Number of columns specified in columns_to_get in a read request	0 to 128	The maximum number of columns obtained from a row of data in a read request cannot exceed 128.
Queries per second (QPS) at the table level	10 QPS	The QPS for tables in an instance cannot exceed 10. For more information about table-level operations, see the "Table operations" section in OperationsSummary.

Operation	Limit	Description
Count of UpdateTable operations for a single table	Tablestore does not impose limits on the count of UpdateTable operations for a single table.	The limit on the count of UpdateTable operations for a single table follows the limit on the frequency of adjustment for a single table.
Frequency of calling the UpdateTable operation for a single table	Once every 2 minutes	The reserved read or write throughput for a single table can be adjusted once every two minutes at most.
Number of rows read by one BatchGetRow request	100	None.
Number of rows written by one BatchWriteRow request	200	None.
Size of data written by one BatchWriteRow request	4 MB	None.
Size of data written by one PutRow request	4 MB	None.
Size of data written by one UpdateRow request	4 MB	None.
Size of data scanned at a time by one GetRange request	5,000 rows or 4 MB	The size of data scanned at a time by one GetRange request cannot exceed 5,000 rows or 4 MB. When one of the limits is exceeded, data that exceeds the limit is truncated at the row level. The primary key information of the next row of data is returned.
Data size of an HTTP request body	5 MB	None.
Number of filters in one read request	10	None.

13.2. Secondary index limits

This topic describes the limits of secondary index.

Limits on index tables

Item	Limit	Description
Length of a table name	1~255 Bytes	The name of a table can contain uppercase and lowercase letters, digits, and hyphens (-). The name must start with a letter or an underscore (_).

Item	Limit	Description
Number of secondary indexes for a single base table	5	A maximum of five index tables can be created for each base table.
Number of indexed columns	1 to 4	A maximum of four indexed columns can be added for an index table. Indexed columns consist of the primary key of the base table and predefined columns. The primary key of an index table consists of indexed columns and autocompleted primary key columns of the base table.
Data types supported by indexed columns	STRING, INTEGER, and BINARY	The data types supported by indexed columns are STRING, INTEGER, and BINARY.
Number of attribute columns	32	A maximum of 32 attribute columns can be added to an index table. Attribute columns of an index table consist of predefined columns of the base table.
Data types supported by attribute columns	STRING, INTEGER, DOUBLE, BOOLEAN, and BINARY	The data types supported by attribute columns are STRING, INTEGER, DOUBLE, BOOLEAN, and BINARY.

Additional limits

Item	Limit	Description
Indexed column	Columns except auto-increment primary key columns	The first primary key column of an index table cannot be an auto-increment column.
Operation on data in index tables	Read-only	You can only read data from index tables. You cannot write data to index tables.
Max versions	Not supported	You cannot create secondary indexes for tables that have max versions enabled.
Time to live (TTL)	Supported	Updates for base tables must be disabled. Make sure that the TTL value set for an index table is consistent with that for the base table.
Stream	Not supported	None.
Query from the base table	Not supported	Queries from the base table are required.

13.3. Search index limits

This topic describes the limits on search indexes.

Mapping

Item	Maximum value	Description
Number of indexed fields	500	The number of fields that can be indexed.
Array length	256	The maximum number of elements in an array.
Number of fields for which EnableSortAndAgg is set to true	100	The number of fields that can be sorted and aggregated.
Number of nested levels	5	Up to five levels can be nested.
Number of child rows in a nested field	256	The maximum number of child rows that are contained in a nested field.
Number of nested fields	25	The number of child fields that can be nested.
Total length of values in all primary key columns	1,000 bytes	The total length of all primary key columns in each row can be a maximum of 1,000 bytes.
Length of the value in the primary key column of the STRING type	1,000 bytes	To index a primary key column of the STRING type, the column value cannot exceed 1,000 bytes in length.
Length of the value in an attribute column of the STRING type if you want to index the column as KEYWORD	4 KB	None.
Length of the value in an attribute column of the STRING type if you want to index the column as TEXT	2 MB	The limit is the same as the length limit on an attribute column in a data table.
Length of a query string that contains a wildcard	32	The query string can be up to 32 characters in length.
Length of a query string that contains a prefix	1,000 bytes	The query string can be up to 1,000 characters in length.

Search

Category	Item	Maximum value	Description
General limits	offset+limit	10000	To increase the number of returned rows, configure the next_token parameter.
	limit	100	• When you call the Search operation to query data of a specified column, the maximum value of the limit parameter can be set to 1000 if the column is contained in search indexes. • To increase the limit, submit a ticket.
	timeout	10s	None.
	CU	100,000	• This limit does not take effect for scanning and analysis requests. • To increase the limit, submit a ticket.
	QPS	100,000	• The upper limit for lightweight transaction processing is 100,000 queries per second (QPS). • To increase the limit, submit a ticket.
	Number of query methods that are specified in a Search call	1024	If complex nested queries are specified in a Search call, query performance is compromised. We recommend that you simplify the queries.
Aggregation	Number of Aggregations at the same level	5	The number of Aggregations is recalculated each time you add a new Aggregation to SubGroupBy.
	Number of GroupBys at the same level	5	The number of GroupBys is recalculated each time you add a new GroupBy to SubGroupBy.
	Number of nested GroupBys	3	The root GroupBy is calculated as a nested level.
	Number of Filters in GroupByFilter	10	None.
	Number of groups returned by GroupByField	2,000	None.
	Number of Ranges in GroupByRange	100	None.

Category	Item	Maximum value	Description
	Number of Ranges in GroupByGeodistance	10	None.

ParallelScan

Category	Item	Description
General limits	offset+limit	When you use parallel scan, you cannot configure the offset and limit parameters. The results that are returned are displayed in chronological order.
	limit	The maximum value is 2,000.
	CU	None.
	QPS	None.
	Maximum number of parallel tasks	The value of the MaxParallel parameter. You can call the ComputeSplits operation to obtain the value of the parameter.

Index

Item	Maximum value	Description
Rate	50,000 rows/s	• The first time when data is written to a table or when a large volume of data is written in a short period of time, Tablestore balances loads within a few minutes. • The maximum rate of indexing TEXT fields is 10,000 rows/s because this process consumes a large number of CPU resources for tokenization. • To increase the limit, submit a ticket.
Synchronization latency	3s	• In most cases, the synchronization latency is within three seconds. • It takes up to one minute to initialize a new index.
Number of rows	50,000,000,000	To increase the limit, submit a ticket.
Total size	50 TB	To increase the limit, submit a ticket.

Other limits

Search indexes are available in the following regions: China (Hangzhou), China (Shanghai), China (Beijing), China (Zhangjiakou), China (Shenzhen), China (Hong Kong), Singapore (Singapore), Australia (Sydney), Indonesia (Jakarta), Japan (Tokyo), Germany (Frankfurt), UK (London), US (Silicon Valley), US (Virginia), India (Mumbai), and Philippines (Manila).

Note To use search indexes in a wider range of regions, submit a ticket. When you submit a ticket, you must specify the limits and the limit values. You must also specify the scenarios in which you want to use the new limits and the requirements based on which you want to use the new limits. The requirements that you specify in the ticket are recorded for future development purposes.

13.4. SQL limits

This topic describes the limits of SQL.

Configuration limits

Notice The database names, table names, and column names cannot be the reserved words or keywords in SQL. For more information about the reserved words and keywords, see Reserved words and keywords.

Item	Limit	Description
Database name length	3 to 16 bytes	The database name corresponds to the instance name. The database name can contain letters, digits, and hyphens (-). The name must start with a letter and cannot end with a hyphen (-).
Table name length	1 to 255 bytes	The table name corresponds to the data table name or index table name. The table name can contain letters, digits, and underscores (_). The name must start with a letter or an underscore (_).
Column name length	1 to 255 bytes	The column name corresponds to the column name in a data table or index table. The column name can contain letters, digits, and underscores (_). The name must start with a letter or an underscore (_).
Number of columns	1 to 32	If your business requirements are not met due to the limit, submit a ticket.
Size of the value in a primary key column of the STRING type	1 KB	The size of the value in a primary key column of the STRING type cannot exceed 1 KB.
Size of the value in an attribute column of the STRING type	2 MB	The size of the value in an attribute column of the STRING type cannot exceed 2 MB.

Item	Limit	Description
Size of the value in a primary key column of the Binary (Blob) type	1 KB	The size of the value in a primary key column of the Binary (Blob) type cannot exceed 1 KB.
Size of the value in an attribute column of the Binary (Blob) type	2 MB	The size of the value in an attribute column of the Binary (Blob) type cannot exceed 2 MB.

Operation limits

Item	Limit	Description
Amount of data for a single scan	128 MB or 100,000 rows	The maximum number of rows for a single scan is 100,000 or the maximum amount of data for a single scan is 128 MB. If the upper limit is exceeded, the system returns an error.
Single execution time	30s	The single execution time is related to the complexity of the SQL statement and the amount of data in the table. The maximum duration is 30 seconds. If the maximum duration is exceeded, the system returns an error.
Data type and position of a column	Unmodifiable	The data type and position of a column cannot be modified.
Case sensitivity	Not case-sensitive	The table names and column names in Tablestore are both case-sensitive. When SQL is used, the Tablestore table names and column names are converted into lowercase letters for matching. In this case, if you want to perform operations on the Aa column in a Tablestore table, you can use AA, aa, aA, or Aa in SQL. Therefore, the table names or column names in Tablestore cannot be AA, aa, aA, and Aa at the same time.

13.5. Limits on the TimeSeries model

This topic describes the limits on the TimeSeries model.

Item	Limit
Name of a time series table	The name of a time series table must be 1 to 128 bytes in size, and can contain letters, digits, and underscores (_). The name cannot start with a digit.
Name of a column in a time series table	The name of a column in a time series table must be 1 to 128 bytes in size, and can contain lowercase letters, digits, and underscores (_). The name cannot contain _m_name, _data_source, _tags, _time, _meta_update_time, or _attributes. The name cannot start with a digit.

Item	Limit
Measurement name	A measurement name must be a UTF-8-encoded string that is 1 to 128 bytes in size. The name cannot contain number signs (#) or non-printable characters such as spaces.
Data source	A data source must be a UTF-8-encoded string that is 0 to 256 bytes in size.
Tags	The tags must be in the ["k1=v1","k2=v2"] format. Each tag consists of a key and a value that are connected by an equal sign (=). The tag key must contain only printable ASCII characters. The tag value can be a UTF-8-encoded string. Both the tag key and tag value cannot contain double quotation marks (") or equal signs (=). The tags cannot exceed 512 bytes in size.
Time column	The value of a time column must be greater than or equal to 0. Unit: microseconds.
Maximum number of columns that can be written at a time	You can write up to 1,024 attribute columns in a single row to a time series table at a time.
Maximum number of rows that can be written at a time	You can write up to 200 rows to a time series table at a time.
Maximum size of data that can be written to a time series table at a time	You can write up to 4 MB of data to a time series table at a time.
Size of the value in a column of the STRING type	The size of the value in a column of the STRING type cannot exceed 2 MB.
Size of the value in a column of the BINARY type	The size of the value in a column of the BINARY type cannot exceed 2 MB.

14.HBase

14.1. Tablestore HBase Client

In addition to SDKs and RESTful operations, you can also use Tablestore HBase Client to access Tablestore. Java applications that support open source HBase operations can use Tablestore HBase Client to access Tablestore.

Based on Tablestore SDKs for Java V4.2.x and later, Tablestore HBase Client supports open source operations for HBase V1.x.x and later.

You can obtain Tablestore HBase Client by using one of the following methods:

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

```
<dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
       </dependency>
   </dependencies>
```

Tablestore is a fully managed NoSQL database service. When you use Tablestore HBase Client, you can ignore HBase Server. Instead, you need only to perform table or data operations by using operations provided by Client.

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

Item	Tablestore	Self-built HBase cluster
Cost	Charges fees based on actual data volumes. Tablestore provides high performance and capacity instances to meet the requirements of different scenarios.	Allocates resources based on traffic peaks. Resources remain idle during off-peak periods, which results in high operation and maintenance costs.
Security	Integrates Alibaba Cloud RAM and supports multiple authentication and authorization mechanisms, VPC, and primary/RAM user management. Authorization granularity can be defined at both the table-level and operation-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%.	Requires extra mechanisms to ensure cluster reliability.

Item	Tablestore	Self-built HBase cluster
Scalability	Server Load Balancer (SLB) of Tablestore 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 and offline processes are required if a cluster reaches high usage capacity, which impacts online services.

14.2. Features of Tablestore HBase Client

This topic describes the features and operations supported by Tablestore HBase Client.

Differences between API operations supported by Tablestore and HBase

As a NoSQL database service, Tablestore hides infrastructure details such as table splitting, Dump, Compact, and Region Server. You need only to pay attention to data usage. Tablestore HBase Client and HBase are similar in terms of data model and features, but they have different operations.

Features supported by Tablestore HBase Client operations

- CreateTable

  Tablestore does not support ColumnFamily and all data can be considered to be in the same ColumnFamily. This means that TTL and Max Versions of Tablestore are at the table-level. Therefore, Tablestore supports the following features:

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

- Put

Features	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	Not supported
ttl	Not supported
Cell Visibility	Not supported
tag	Not supported

- Get

  Tablestore guarantees high data consistency. If the HTTP 200 status code (OK) is returned after you perform a write operation, the data is permanently written to all copies, and can be immediately read by Get.

Features	Supported or Not
Reads a row	Supported
Reads all columns in a ColumnFamily	Supported
Reads data from a specified column	Supported
Reads data that has a specified timestamp	Supported
Reads data of a specified number of versions	Supported
TimeRange	Supported
ColumnfamilyTimeRange	Not supported
RowOffsetPerColumnFamily	Supported
MaxResultsPerColumnFamily	Not supported
checkExistenceOnly	Not supported
closestRowBefore	Supported
attribute	Not supported
cacheblock:true	Supported
cacheblock:false	Not supported

Features	Supported or Not
IsolationLevel:READ_COMMITTED	Supported
IsolationLevel:READ_UNCOMMITTED	Not supported
IsolationLevel:STRONG	Supported
IsolationLevel:TIMELINE	Not supported

- Scan

  Tablestore guarantees high data consistency. If the HTTP 200 status code (OK) is returned after you perform a write operation, the data is permanently written to all copies, which can be immediately read by Scan.

Features	Supported or Not
Determines a scanning range based on the specified start and stop	Supported
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	Not supported
maxResultSize, which indicates the maximum size of the returned data volume	Not supported
small	Not supported
batch	Not supported
cacheblock:true	Supported
cacheblock:false	Not supported
IsolationLevel:READ_COMMITTED	Supported
IsolationLevel:READ_UNCOMMITTED	Not supported
IsolationLevel:STRONG	Supported
IsolationLevel:TIMELINE	Not supported
allowPartialResults	Not supported

- Batch

Features	Supported or Not
Get	Supported
Put	Supported
Delete	Supported
batchCallback	Not supported

- Delete

Features	Supported or Not
Deletes a row	Supported
Deletes all versions of the specified column	Supported
Deletes the specified version of the specified column	Supported
Deletes the specified ColumnFamily	Not supported
Specifies a timestamp to delete the versions that are equal to the timestamp	Supported
Specifies a timestamp and use deleteFamily and deleteColumns to delete the versions that are earlier than or equal to the timestamp	Not supported
Uses deleteColumn to delete the latest version without specifying a timestamp	Not supported
Uses deleteFamily and deleteColumns to delete the version of the current system time without specifying a timestamp	Not supported
addDeleteMarker	Not supported

- checkAndXXX

Features	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

Features	Supported or Not
Uses the default value if no value is specified	Supported
Checks row A and executes row B.	Not supported

- exist

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

- Filter

Features	Supported or Not
ColumnPaginationFilter	columnOffset and count not supported
SingleColumnValueFilter	Supported: LongComparator, BinaryComparator, and ByteArrayComparable Not supported: RegexStringComparator, SubstringComparator, and BitComparator

Operations not supported by Tablestore HBase Client

- Namespaces

  Tablestore uses instances to manage a data table. An instance is the minimum billing unit in Tablestore. You can manage instances in the Tablestore 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

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

- Snapshots

  Tablestore does not support Snapshots, or related features of Snapshots.

- Table management

Tablestore automatically splits, merges, and compacts 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

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

  - coprocessorService()
  - coprocessorService(ServerName serverName)
  - getMasterCoprocessors()

- Distributed procedures

  Tablestore 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

  Tablestore does not support atomic increase and decrease or atomic Append.

14.3. Differences between Tablestore and HBase

Tablestore HBase Client is used in a similar but differentiated way to HBase. This topic introduces the features of Tablestore HBase Client.

Table

Tablestore supports only single ColumnFamilies.

Row and Cell

- Tablestore does not support ACL settings.
- Tablestore does not support Cell Visibility settings.
- Tablestore does not support Tag settings.

GET

Tablestore supports only single ColumnFamilies. Therefore, Tablestore does not support ColumnFamily-related operations, including:

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

SCAN

Similar to GET, Tablestore does not support ColumnFamily-related operations and cannot be used to set partial optimization operations, including:

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

Batch

Tablestore does not support BatchCallback.

Mutations and Deletions

- Tablestore does not support the deletion of the specified ColumnFamily.
- Tablestore does not support the deletion of the versions that has the latest timestamp.
- Tablestore does not support the deletion of all versions earlier than the specified timestamp.

Increment and Append

Tablestore does not support Increment or Append.

Filter

- Tablestore supports ColumnPaginationFilter.
- Tablestore supports FilterList.
- Tablestore partially supports SingleColumnValueFilter, and supports only BinaryComparator.
- Tablestore does not support other filters.

Optimization

Some of the HBase operations involve access and storage optimization. The following operations are not opened in Tablestore:

- 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` operations of HBase are used for management and control, most of which are not required in Tablestore.

Tablestore is a cloud service that automatically performs operations such as operation and maintenance, management, and control, which you do not need to concern about. Tablestore does not support a few of other operations.

- CreateTable

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

- Maintenance task

  In Tablestore, the following operations 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 Tablestore, the instance name is similar to Namespaces in HBase. Therefore, Tablestore does not support Namespaces-related operations, including:

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

- Region

  Tablestore automatically performs Region-related operations. Therefore, it does not support the following operations:

  - 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

Tablestore does not support Snapshots-related operations.

Replication

Tablestore does not support Replication-related operations.

Coprocessors

Tablestore does not support Coprocessors-related operations.

Distributed procedures

Tablestore does not support Distributed procedures-related operations.

Table management

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

- 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)

Limits

Tablestore is a cloud service. To guarantee the optimal overall performance, some parameters are limited and cannot be reconfigured. For more information about the limits, see General limits.

14.4. Migrate data from HBase to Tablestore

Tablestore HBase Client is encapsulated based on HBase Client, and is used in a similar but differentiated way as HBase Client. This topic describes how to migrate data from HBase Client to Tablestore HBase Client.

Dependencies

Tablestore HBase Client V1.2.0 depends on HBase Client 1.2.0 and Tablestore SDK for Java V4.2.1. The following code provides an example on how to configure pom.xml:

```
<dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
       </dependency>
   </dependencies>
```

If you want to use other version of HBase Client or Tablestore SDK for Java, you can use the exclusion tag. In the following example, HBase Client 1.2.1 and Tablestore V4.2.0 are used.

```
<dependencies>
     <dependency>
         <groupId>com.aliyun.openservices</groupId>
         <artifactId>tablestore-hbase-client</artifactId>
         <version>1.2.0</version>
         <exclusions>
             <exclusion>
                 <groupId>com.aliyun.openservices</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.openservices</groupId>
         <artifactId>tablestore</artifactId>
         <classifier>jar-with-dependencies</classifier>
         <version>4.2.0</version>
     </dependency>
 </dependencies>
```

Tablestore HBase Client V1.2.x is compatible with only HBase Client 1.2.x because HBase Client 1.2.x has different operations from other versions.

If you want to use HBase Client 1.1.x, you must use Tablestore HBase Client V1.1.x.

If you want to use HBase Client 0.x.x, see Make Tablestore HBase Client compatible with HBase versions earlier than 1.0.0.

Configure files

To migrate data from HBase Client to Tablestore HBase Client, you must modify the following parameters in the configuration file:

- HBase Connection type

  Set Connection to TablestoreConnection.

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

- Configuration items of Tablestore

Tablestore is a cloud service that enables strict permission management. To access Tablestore, you must configure authentication information such as AccessKey pairs.

- You must configure the following items before you can access Tablestore:

```
<property>
           <name>tablestore.client.endpoint</name>
           <value></value>
       </property>
       <property>
           <name>tablestore.client.instancename</name>
           <value></value>
       </property>
       <property>
           <name>tablestore.client.accesskeyid</name>
           <value></value>
       </property>
       <property>
           <name>tablestore.client.accesskeysecret</name>
           <value></value>
       </property>
```

- The following items are optional:

```
       <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.connections</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
  - Tablestore supports only single ColumnFamilies. When you use HBase operations, you must set the content of the family.

    `hbase.client.tablestore.family` indicates the global configuration, while `hbase.client.tablestore.family.$tablename` indicates the configurations of a single table.
  - Rule: For a table named T, search for `hbase.client.tablestore.family.T`. 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

  The maximum number of connections. Default value: 300.
- tablestore.client.socket.timeout

  The timeout period of the socket. Default value: 15. Unit: seconds.
- tablestore.client.connection.timeout

  The connection timeout period. Default value: 15. Unit: seconds.
- tablestore.client.operation.timeout

  The timeout period to access an operation. The default value is Integer.MAX_VALUE, which indicates that the operation never times out.
- tablestore.client.retries

  The number of retry attempts when a request fails. Default value: 3.

14.5. Make Tablestore HBase Client compatible with HBase versions earlier than 1.0

Tablestore HBase Client supports the operations of HBase 1.0.0 and later. This topic describes how to make Tablestore HBase Client compatible with the operations of HBase versions earlier than 1.0.

Compared with earlier versions, HBase 1.0.0 has some major changes, which are incompatible with HBase of earlier versions.

This topic also describes the major changes to facilitate your operations.

Connections

HConnection is deprecated in HBase 1.0.0 and later. We recommend that you use `org.apache.hadoop.hbase.client.ConnectionFactory` to create a class to implement Connections, and replace the deprecated ConnectionManager and HConnectionManager with ConnectionFactory.

Connection creation is a heavy-weight operation. Connection implementations are thread-safe. You can create a connection, and share it with different threads.

For HBase 1.0.0 and later, you must manage the lifecycle of the connection and close the connection after use.

The following code is the latest code used to create a connection:

```
Connection connection = ConnectionFactory.createConnection(config);
// ...
connection.close();
```

TableName class

If you use HBase versions earlier than 1.0.0, you can specify a STRING-type table name when you create a table. For HBase 1.0.0 and later, you must use the `org.apache.hadoop.hbase.TableName` class.

The following code is the latest code used to specify a STRING-type table name:

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

Table, BufferedMutator, and RegionLocator operations

The HTable operation is replaced by the Table, BufferedMutator, and RegionLocator operations in HBase 1.0.0 and later.

- `org.apache.hadoop.hbase.client.Table` : performs read and write operations on a single table.
- `org.apache.hadoop.hbase.client.BufferedMutator` : writes data asynchronously. This operation corresponds to `setAutoFlush(boolean)` of the HTableInterface operation of the earlier HBase versions.
- `org.apache.hadoop.hbase.client.RegionLocator` : indicates the partition information of the table.

The Table, BufferedMutator, and RegionLocator operations are not thread-safe. However, they are lightweight and can be used to create an object for each thread.

Admin operations

The HBaseAdmin operation is replaced by `org.apache.hadoop.hbase.client.Admin` in HBase 1.0.0 and later. Tablestore is a cloud service, and most Tablestore O&M operations are automatically processed. Therefore, most Admin operations are not supported by Tablestore. For more information, see Differences between Tablestore and HBase.

Use a connection instance to create an admin instance:

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

14.6. Quick start

This topic describes how to use Tablestore HBase Client to implement a simple program.

> **Note** This sample program uses HBase operations to access Tablestore. The complete sample program is stored in the Aliyun Tablestore HBase client for Java project in GitHub. The directory of the sample program is *src/test/java/samples/HelloWorld.java*.

Procedure

1. Configure project dependencies

Configure the following Maven dependencies:

```
   <dependencies>
        <dependency>
            <groupId>com.aliyun.openservices</groupId>
            <artifactId>tablestore-hbase-client</artifactId>
            <version>1.2.0</version>
        </dependency>
    </dependencies>
```

For more information about dependency configurations, see Migrate data from HBase to Tablestore.

2. Configure files

   Add the following configurations to hbase-site.xml:

```
<configuration>
    <property>
        <name>hbase.client.connection.impl</name>
        <value>com.alicloud.tablestore.hbase.TablestoreConnection</value>
    </property>
    <property>
        <name>tablestore.client.endpoint</name>
        <value>endpoint</value>
    </property>
    <property>
        <name>tablestore.client.instancename</name>
        <value>instance_name</value>
    </property>
    <property>
        <name>tablestore.client.accesskeyid</name>
        <value>access_key_id</value>
    </property>
    <property>
        <name>tablestore.client.accesskeysecret</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_adaptor</value>
    </property>
</configuration>
```

   For more information about configurations, see Migrate data from HBase to Tablestore.

3. Connect to Tablestore

   Create a TableStoreConnection object to connect to Tablestore.

```
        Configuration config = HBaseConfiguration.create();
        // Create a Tablestore Connection
        Connection connection = ConnectionFactory.createConnection(config);
        // Admin is used for creation, management, and deletion
        Admin admin = connection.getAdmin();
```

4. Create a table

   Create a table by using the specified table name. Use the default MaxVersion and TimeToLive values.

```
        // Create an HTableDescriptor, which contains only one ColumnFamily.
        HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf(TABLE_NAME
));
        // Create a ColumnFamily. Use the default MaxVersion and TimeToLive values. The
default value of MaxVersion is 1. The default value of TimeToLive is Integer.INF_MAX.
        descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));
        // Use the createTable operation of the Admin to create a table.
        System.out.println("Create table " + descriptor.getNameAsString());
        admin.createTable(descriptor);
```

5. Write data

   The following code provides an example on how to write a row of data to Tablestore.

```
        // Create a TablestoreTable to perform operations such as read, write, update,
and deletion on a single table.
        Table table = connection.getTable(TableName.valueOf(TABLE_NAME));
        // Create a Put object and use row_1 as the primary key.
        System.out.println("Write one row to the table");
        Put put = new Put(ROW_KEY);
        // Add a column. Tablestore supports only single ColumnFamilies. The ColumnFami
ly name is configured in hbase-site.xml. If the ColumnFamily name is not configured, th
e default name "f" is used. The value of COLUMN_FAMILY_NAME may be null when data is wr
itten.
        put.addColumn(COLUMN_FAMILY_NAME, COLUMN_NAME, COLUMN_VALUE);
        // Run put for Table, and use HBase operations to write the row of data to Tabl
estore
        table.put(put);
```

6. Read data

   The following code provides an example on how to read data of a 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);
        // Display the results.
        String value = Bytes.toString(getResult.getValue(COLUMN_FAMILY_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);
```

7. Scan data

   The following code provides an example on how to read data within a specified range:

```
    Scan data of all rows in the table.
    System.out.println("Scan for all rows:");
    Scan scan = new Scan();
    ResultScanner scanner = table.getScanner(scan);
    // Print the results cyclically.
    for (Result row : scanner) {
        byte[] valueBytes = row.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME);
        System.out.println('\t' + Bytes.toString(valueBytes));
    }
```

8. Delete a table

   The following code provides an example on how to use Admin operations to delete a table.

```
        print("Delete the table");
        admin.disableTable(table.getName());
        admin.deleteTable(table.getName());
```

Complete code

```
package samples;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
import java.io.IOException;
public class HelloWorld {
    private static final byte[] TABLE_NAME = Bytes.toBytes("HelloTablestore");
    private static final byte[] ROW_KEY = Bytes.toBytes("row_1");
    private static final byte[] COLUMN_FAMILY_NAME = Bytes.toBytes("f");
    private static final byte[] COLUMN_NAME = Bytes.toBytes("col_1");
    private static final byte[] COLUMN_VALUE = Bytes.toBytes("col_value");
    public static void main(String[] args) {
        helloWorld();
    }
    private static void helloWorld() {
        try  {
            Configuration config = HBaseConfiguration.create();
            Connection connection = ConnectionFactory.createConnection(config);
            Admin admin = connection.getAdmin();
            HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf(TABLE_NAME
));
            descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));
            System.out.println("Create table " + descriptor.getNameAsString());
            admin.createTable(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_FAMILY_NAME, COLUMN_NAME, COLUMN_VALUE);
            table.put(put);
            Result getResult = table.get(new Get(ROW_KEY));
            String value = Bytes.toString(getResult.getValue(COLUMN_FAMILY_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:");
            ResultScanner scanner = table.getScanner(scan);
            for (Result row : scanner) {
                byte[] valueBytes = row.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME);
                System.out.println('\t' + Bytes.toString(valueBytes));
            }
            System.out.println("Delete the table");
            admin.disableTable(table.getName());
            admin.deleteTable(table.getName());
            table.close();
            admin.close();
            connection.close();
        } catch (IOException e) {
            System.err.println("Exception while running HelloTablestore: " + e.toString());
            System.exit(1);
        }
    }
}
```

15.Authorization management

15.1. RAM and STS

The permission management mechanism of Alibaba Cloud includes Resource Access Management (RAM) and Security Token Service (STS). RAM user accounts with different permissions can be created to access Tablestore, and temporary access permission can also be granted to RAM users. RAM and STS greatly improve management flexibility and security.

RAM is used to control the permissions of each account. RAM allows you to manage permissions by granting different permissions to different RAM user accounts created under Alibaba Cloud accounts. For more information, see RAM documentation.

STS is a security credential (token) management system that grants users temporary access permission. For more information, see STS.

Background

RAM and STS enable you to securely grant permissions to users without exposing your Alibaba Cloud account AccessKey pair. If the AccessKey pair of your Alibaba Cloud account is leaked, other users can operate on the resources under the account and access important information.

RAM allows you to manage permissions granted to RAM users on different entities and minimizes the adverse impact if the AccessKey pair of a RAM user is leaked. RAM user accounts are often used long term to perform&nbsp;operations. To ensure account confidential, the AccessKey pairs of RAM user accounts must be kept confidential.

In contrast to the permanent permission management function provided by RAM, STS provides temporary access authorization through a temporary AccessKey pair and token to allow temporary access to Tablestore. The permissions obtained from STS are restricted and are only valid for a limited period of time to minimize the adverse impact on the system in case of information leakage.

Terms

The following table describes terms related to RAM and STS.

Term	Description
RAM user account	RAM user accounts are created under an Alibaba Cloud account and assigned independent passwords and permissions. Each RAM user account has an AccessKey pair. RAM user accounts can be used to perform authorized operations in the same way as the Alibaba Cloud account. In most cases, a RAM user account can be considered as a user with certain permissions or an operator with permissions for specific operations.
role	A role is a set of permissions that a user can assume. Roles do not have independent logon passwords and AccessKey pairs. RAM user accounts can assume roles. Permissions of a role are granted to RAM user accounts that assume the role.

Term	Description
policy	Policies are rules used to define permissions, such as the permissions to read from or write to certain resources.
resource	Resources are the cloud resources that users can access, such as individual Tablestore instances, all Tablestore instances, or a certain table in an instance.

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

Example:

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

To maintain the security of the Alibaba Cloud account, Alice uses RAM to create two RAM user accounts: Bob and Carol. Bob has read and write permissions on alice_a, and Carol has read and write permissions on alice_b. Bob and Carol both have their own AccessKey pairs. If the AccessKey pair of Bob or Carol is leaked, only the corresponding instance is affected. Alice can then revoke the permissions of the compromised RAM user account through the console.

If Alice needs to authorize another RAM user to read the tables in alice_a, instead of disclosing Bob's AccessKey pair to the user, Alice can create a new role such as AliceAReader and grant that role the read permission on alice_a. However, AliceAReader cannot be used directly because it does not have a corresponding AccessKey pair.

To obtain temporary authorization, Alice can call AssumeRole to inform STS that the RAM user account Bob wants to assume the AliceAReader role. If AssumeRole is successfully called, STS returns a temporary AccessKey ID, AccessKey secret, and security token as access credentials. A temporary user assigned with these credentials is authorized to temporarily access alice_a. The expiration time of the credentials is specified when AssumeRole is called.

Design philosophy behind RAM and STS

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

RAM user accounts and roles are separated to keep the entity that performs operations separating from the virtual entity that represents a group of permissions. Assume that a user requires multiple permissions such as read and write permissions, but each operation only requires one of the permissions. In this case, you can create two roles: one with the read permission and the other one with the write permission. Then you can create a RAM user account that does not have any permissions but can assume these roles. When the user needs to read or write data, the RAM user account can temporarily assume the role with the required permission. In addition, roles can be used to grant permissions to other Alibaba Cloud users, which makes collaborations easier and maintains strict account security.

Flexible access control does not mean that you have to use all these functions. You may only use a subset of functions as needed. For example, if you do not need to use temporary access credentials that have an expiration time, you can use only the RAM user account function.

15.2. Configure user permissions

The permission management mechanism of Alibaba Cloud includes Resource Access Management (RAM) and Security Token Service (STS). RAM users that have different permissions can access Tablestore. The permission management mechanism also authorizes STS. RAM and Security Token Service (STS) make management more flexible and secure.

Background information

RAM and STS enable you to grant permissions without exposing the AccessKey pair of your Alibaba Cloud account. If the AccessKey pair of the Alibaba Cloud account is leaked, other users can perform operations on all the resources of the Alibaba Cloud account and steal important information.

- RAM is a service provided by Alibaba Cloud. RAM allows you to manage user identities and resource access permissions.

  RAM allows you to create and manage multiple identities in an Alibaba Cloud account, and grant different permissions to a single identity or a group of identities. In this way, you can authorize different identities to access different Alibaba Cloud resources. For more information, see What is RAM?.

- STS allows you to manage temporary access from other users to your Alibaba Cloud resources.

  You can use STS to grant temporary access tokens to RAM entities such as RAM users and RAM roles. You can customize the validity period and access permissions of these STS tokens. For more information, see What is STS?.

RAM is an access control service that provides long-term permission management mechanism. The owner of an Alibaba Cloud account can create RAM users and grant different permissions to the RAM users. This way, if an AccessKey pair of a RAM user is disclosed, the information that is leaked is limited. RAM users remain valid for a long period of time. The AccessKey pairs of RAM users must be kept confidential.

In contrast to the long-term permission management mechanism provided by RAM, STS provides temporary access authorization by using a temporary AccessKey pair and token to allow temporary access to Tablestore. Permissions obtained from STS are strictly restricted and have time limits. Therefore, even if information is disclosed, your system is not severely affected.

Grant permissions to a RAM user

1. Create a RAM user. For more information, see Create a RAM user.
2. Grant permissions to the RAM user. For more information, see Grant permissions to a RAM user.
   - To manage Tablestore such as creating an instance, grant the AliyunOTSFullAccess permission to the RAM user.
   - If the RAM user requires read-only access to Tablestore such as reading data from a table, grant the AliyunOTSReadOnlyAccess permission to the RAM user.
   - If the RAM user requires write-only access to Tablestore such as creating a table, grant the AliyunOTSWriteOnlyAccess permission to the RAM user.

   > **Note** For more information about how to implement finer-grained permission control and configure policies, see Create a custom policy. For more information, see Custom permissions.

3. Enable a multi-factor authentication (MFA) device for a RAM user. For more information, see Enable an MFA device for a RAM user.

Grant permissions to a temporary user

1. Create a temporary role and grant permissions.

   i. Create a RAM role for a trusted Alibaba Cloud account. For more information, see Create a RAM role for a trusted Alibaba Cloud account.

      Create two roles named RamTestAppReadOnly and RamTestAppWrite. RamTestAppReadOnly is used to read data, and RamTestAppWrite is used to upload files.

   ii. Create a custom policy. For more information, see Create a custom policy.

      > **Note** To implement finer-grained permission control, you can customize the permissions of a policy. For more information, see Custom permissions.

      Create two policies named ram-test-app-readonly and ram-test-app-write.

      ▪ Ram-test-app-readonly

```
{
"Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ots:BatchGet*",
        "ots:Describe*",
        "ots:Get*",
        "ots:List*"
      ],
      "Resource": [
        "acs:ots:*:*:instance/ram-test-app",
        "acs:ots:*:*:instance/ram-test-app/table/*"
      ]
    }
],
"Version": "1"
}
```

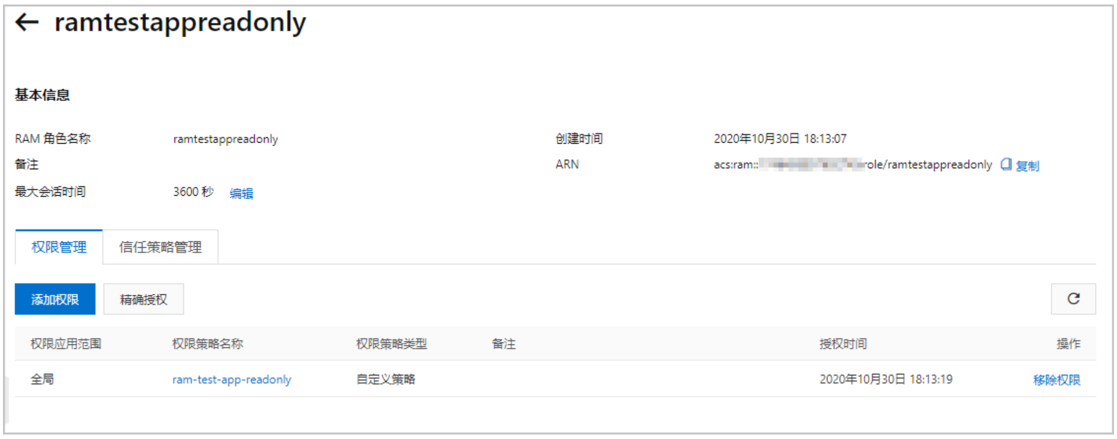
- ram-test-app-write

```
{
"Statement": [
       {
         "Effect": "Allow",
         "Action": [
           "ots:Create*",
           "ots:Insert*",
           "ots:Put*",
           "ots:Update*",
           "ots:Delete*",
           "ots:BatchWrite*"
         ],
         "Resource": [
           "acs:ots:*:*:instance/ram-test-app",
           "acs:ots:*:*:instance/ram-test-app/table/*"
         ]
       }
],
"Version": "1"
}
```

iii. Grant permissions to a temporary role. For more information, see Grant permissions to a RAM role.

Assign the ram-test-app-readonly policy to RamTestAppReadOnly. These are read-only permissions on Tablestore. Assign the ram-test-app-write policy to RamTestAppWrite. These are write-only permissions on Tablestore.

After you complete the authorization, record the **ARN** of the role. ARN indicates the ID of the role that the RAM user assumes. The following figure shows the ARN.



ramtestappreadonly

基本信息

RAM 角色名称 ramtestappreadonly

备注

最大会话时间 3600 秒

创建时间 2020年10月30日 18:13:07

ARN acs:ram::[redacted]role/ramtestappreadonly

权限应用范围	权限策略名称	权限策略类型	备注	授权时间	操作
全局	ram-test-app-readonly	自定义策略		2020年10月30日 18:13:19	移除权限

2. Grant temporary access permissions.

i. Create a custom policy. For more information, see Create a custom policy.

> Note To implement finer-grained permission control, you can customize the permissions of a policy. For more information, see Custom permissions.

Create two policies named AliyunSTSAssumeRolePolicy 2016011401 and AliyunSTSAssumeRolePolicy 2016011402. Resource indicates the ARN of the role.

- AliyunSTSAssumeRolePolicy2016011401

```
{
"Version": "1",
"Statement": [
    {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "acs:ram:198***237:role/ramtestappreadonly"
    }
]
}
```

- AliyunSTSAssumeRolePolicy2016011402

```
{
"Version": "1",
"Statement": [
    {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "acs:ram:198***237:role/ramtestappwrite"
    }
]
}
```

ii. Grant permissions to the temporary role that the RAM user assumes. For more information, see Grant permissions to a RAM user.

Grant the custom AliyunSTSAssumeRolePolicy2016011401 and AliyunSTSAssumeRolePolicy2016011402 policies to the RAM user named ram_test_app.

3. Obtain temporary access credentials from STS. For more information, see AssumeRole.
4. Use temporary permissions to read and write data.

   You can use the temporary permission to call the SDKs of different programming languages to access Tablestore. You can use the following method to create an OTSClient object and add parameters obtained from STS such as AccessKeyId, AccessKeySecret, and SecurityToken:

```
OTSClient client = new OTSClient(otsEndpoint, stsAccessKeyId, stsAccessKeySecret, insta
nceName, stsToken);
```

15.3. AliyunServiceRoleForOTSDataDelivery role

Before you use the data delivery feature of Tablestore, you must have the permissions to access Object Storage Service (OSS) resources. To grant the permissions to access OSS resources to you, the system automatically creates the Tablestore service-linked role AliyunServiceRoleForOTSDataDelivery in the Tablestore console.

> Note
>
> For more information about service-linked roles, see Service linked roles.

Create the service-linked role

Before you use the data delivery feature of Tablestore, the system automatically creates the Tablestore service-linked role AliyunServiceRoleForOTSDataDelivery in the Tablestore console.

The permission policy for AliyunServiceRoleForOTSDataDelivery is AliyunServiceRolePolicyForOTSDataDelivery. The following operations on OSS resources are supported: PutObject, AbortMultipartUpload, PutObjectTagging, GetObject, and DeleteObjectTagging.

Delete the service-linked role

Before you delete the service-linked role AliyunServiceRoleForOTSDataDelivery, make sure that data delivery is not in use for all instances in the current account.

> Notice
>
> After you delete the Tablestore service-linked role, data in the current account cannot be delivered to OSS.

To delete the service-linked role, perform the following steps:

1. Log on to the RAM console.
2. In the left-side navigation pane, choose **Identities** > **Roles**.
3. On the **Roles** page, enter AliyunServiceRoleForOTSDataDelivery in the search box. The AliyunServiceRoleForOTSDataDelivery role is displayed.
4. Click **Delete** in the **Actions** column.
5. In the message that appears, click **OK**.
    - If data delivery is in use for instances in the current account, you cannot delete the AliyunServiceRoleForOTSDataDelivery role. You must delete the delivery tasks from the instances before you can delete the role.
    - If no instances in the current account are using data delivery, you can delete the AliyunServiceRoleForOTSDataDelivery role.

FAQ

Why is the system unable to create the Tablestore service-linked role AliyunServiceRoleForOTSDataDelivery for a RAM user?

The system creates the Tablestore service-linked role only for users that have the required permissions. If the Tablestore service-linked role cannot be automatically created for a RAM user, you must attach the following policy to the RAM user.

Replace *The ID of the Alibaba Cloud account* with the ID of your Alibaba Cloud account.

```
{
    "Statement": [
        {
            "Action": [
                "ram:CreateServiceLinkedRole"
            ],
            "Resource": "acs:ram:*:
The ID of the Alibaba Cloud account.
:role/*",
            "Effect": "Allow",
            "Condition": {
                "StringEquals": {
                    "ram:ServiceName": [
                        "arms.aliyuncs.com"
                    ]
                }
            }
        }
    ],
    "Version": "1"
}
```

15.4. Custom permissions

This topic describes the Action, Resource, and Condition parameters and the scenarios for which the parameters are suitable.

Action

The Action parameter defines the specific API operation or operations to allow or deny. When you create an authorization policy for Tablestore, add the `ots:` prefix to each API operation and separate multiple API operations with commas (,). When you configure the Action parameter, you can use the wildcard character (*) for prefix matching and suffix matching.

You can use the Action parameter for the following operations:

- Single API operation

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

- Multiple API operations

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

- All read-only API operations

```
{
  "Version": "1",
  "Statement": [
    {
      "Action": [
        "ots:BatchGet*",
        "ots:Describe*",
        "ots:Get*",
        "ots:List*",
        "ots:Consume*",
        "ots:Search",
        "ots:ComputeSplitPointsBySize"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```

- All read and write API operations

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

- All API operations in SQL

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

Resource

The Resource parameter in Tablestore consists of multiple fields including the service, region, user_id, instance_name, and table_name. Each field supports the wildcard character (*) for prefix matching and suffix matching. You can configure the Resource parameter based on the following format:

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

The fields that are enclosed in brackets are variables. You must set the service field to ots. The value of the region field specifies the region ID, such as cn-hangzhou. The value of the user_id field specifies the ID of your Alibaba Cloud account.

Note The names of Tablestore instances are not case-sensitive. The value of the instance_name field in the Resource parameter must be specified in lower case.

The Resource parameter in Tunnel Service is defined by instance rather than table and includes fields such as the service, region, user_id, and instance_name. You can configure the Resource parameter in Tunnel Service based on the following format:

```
acs:ots:[region]:[user_id]:instance/[instance_name]
```

Resource has the following definitions:

- All resources of users in all regions

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

- All instances and their tables of User 123456 in the China (Hangzhou) region

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

- Instance abc and its tables of User 123456 in the China (Hangzhou) region

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

- All instances whose names contain the prefix abc and their tables

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

- All tables whose names contain the prefix xyz in the instances whose names contain the prefix abc. Instance resources are not included. acs:ots:*:*:instance/abc* does not match this definition.

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

- All instances whose names contain the suffix abc and their tables whose names contain the suffix xyz

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

Tablestore API operations

Tablestore provides the following types of API operations:

- Instance management operations
- Table operations and data read/write operations
- Operations for Tunnel Service

The following tables describe the API operations.

- Instance management operations

  Instance management operations are instance-based operations and can be called only in the console. If you configure the Action and Resource parameters for instance management operations, some console features may be unavailable. The `acs:ots:[region]:[user_id]:` omitted from the names of the following resources. Only the instance and table are described.

API operation/Action	Resource
ListInstance	instance/*

API operation/Action	Resource
InsertInstance	instance/[instance_name]
GetInstance	instance/[instance_name]
DeleteInstance	instance/[instance_name]

- Table operations and data read/write operations

  Table operations and data read/write operations are performed on tables and rows. You can call table operations and data read/write operations by using the Tablestore console or Tablestore SDKs. If you configure the Action and Resource parameters for table operations and data read/write operations, some console features may be unavailable. The `acs:ots:[region]:[user_id]:` prefix is omitted from the names of the following resources. Only the instance and table are described.

API operation/Action	Resource
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]
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]
ComputeSplitPointsBySize	instance/[instance_name]/table/[table_name]
StartLocalTransaction	instance/[instance_name]/table/[table_name]
CommitTransaction	instance/[instance_name]/table/[table_name]
AbortTransaction	instance/[instance_name]/table/[table_name]
CreateIndex	instance/[instance_name]/table/[table_name]
DropIndex	instance/[instance_name]/table/[table_name]

API operation/Action	Resource
CreateSearchIndex	instance/[instance_name]/table/[table_name]
DeleteSearchIndex	instance/[instance_name]/table/[table_name]
ListSearchIndex	instance/[instance_name]/table/[table_name]
DescribeSearchIndex	instance/[instance_name]/table/[table_name]
Search	instance/[instance_name]/table/[table_name]
CreateTunnel	instance/[instance_name]/table/[table_name]
DeleteTunnel	instance/[instance_name]/table/[table_name]
ListTunnel	instance/[instance_name]/table/[table_name]
DescribeTunnel	instance/[instance_name]/table/[table_name]
ConsumeTunnel	instance/[instance_name]/table/[table_name]
BulkImport	instance/[instance_name]/table/[table_name]
BulkExport	instance/[instance_name]/table/[table_name]
SQL_Select	instance/[instance_name]/table/[table_name]
SQL_Create	instance/[instance_name]/table/[table_name]
SQL_DropMapping	instance/[instance_name]/table/[table_name]

- Operations for Tunnel Service

  Operations for Tunnel Service are tunnel-related operations and can be called by using the console or Tablestore SDKs. If you configure the Action and Resource parameters for operations for Tunnel Service, some console features may be unavailable. The `acs:ots:[region]:[user_id]:` prefix is omitted from the names of the following resources. Only the instance and table are described.

API operation/Action	Resource
ListTable	instance/[instance_name]
CreateTable	instance/[instance_name]
UpdateTable	instance/[instance_name]
DescribeTable	instance/[instance_name]
DeleteTable	instance/[instance_name]
GetRow	instance/[instance_name]
PutRow	instance/[instance_name]

API operation/Action	Resource
UpdateRow	instance/[instance_name]
DeleteRow	instance/[instance_name]
GetRange	instance/[instance_name]
BatchGetRow	instance/[instance_name]
BatchWriteRow	instance/[instance_name]
ComputeSplitPointsBySize	instance/[instance_name]
StartLocalTransaction	instance/[instance_name]
CommitTransaction	instance/[instance_name]
AbortTransaction	instance/[instance_name]
CreateIndex	instance/[instance_name]
DropIndex	instance/[instance_name]
CreateSearchIndex	instance/[instance_name]
DeleteSearchIndex	instance/[instance_name]
ListSearchIndex	instance/[instance_name]
DescribeSearchIndex	instance/[instance_name]
Search	instance/[instance_name]
CreateTunnel	instance/[instance_name]
DeleteTunnel	instance/[instance_name]
ListTunnel	instance/[instance_name]
DescribeTunnel	instance/[instance_name]
ConsumeTunnel	instance/[instance_name]

- Instructions
  - You can verify the Action and Resource parameters that are configured for a policy by strings. When you configure the Action and Resource parameters, you can use the wildcard character (*) for prefix matching and suffix matching. If Resource is defined as acs:ots:*:*:instance/*/, acs:ots:*:*:instance/abc cannot match the definition. If Resource is defined as acs:ots:*:*:instance/abc, acs:ots:*:*:instance/abc/table/xyz cannot match the definition.
  - To manage instance resources as a RAM user in the Tablestore console, you must grant the RAM user the read permissions on acs:ots:[region]:[user_id]:instance/* to allow the console to obtain the instance list.

  - For batch API operations, such as BatchGetRow and BatchWriteRow, the backend service authenticates each table that you want to access. Operations can be performed only when all tables are authenticated. Otherwise, an error message is returned.

Condition

Policies can support various authentication conditions, including IP address-based access control, HTTPS-based access control, Multi-Factor Authentication (MFA)-based access control, and time-based access control. All Tablestore API operations support these conditions.

- IP address-based access control

  Resource Access Management (RAM) allows you to specify IP addresses or CIDR blocks that are allowed/restricted to access Tablestore resources. IP address-based access control is suitable for the following scenarios:

  - Allow access from multiple IP addresses. The following sample code allows access from only IP addresses 10.101.168.111 and 10.101.169.111:

    ```
    {
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "ots:*",
            "Resource": "acs:ots:*:*:*",
            "Condition": {
                "IpAddress": {
                    "acs:SourceIp": [
                        "10.101.168.111",
                        "10.101.169.111"
                    ]
                }
            }
        }
    ],
    "Version": "1"
    }
    ```

  - Allow access only from one IP address or CIDR block. The following sample code allows access from only the IP address 10.101.168.111 or the CIDR block 10.101.169.111/24:

    ```
    {
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "ots:*",
            "Resource": "acs:ots:*:*:*",
            "Condition": {
                "IpAddress": {
                    "acs:SourceIp": [
                        "10.101.168.111",
                        "10.101.169.111/24"
                    ]
                }
            }
        }
    ],
    "Version": "1"
    }
    ```

- HTTPS-based access control

  RAM allows you to specify whether requests that are sent over HTTPS can access Tablestore resources.

  The following sample code only allows HTTPS requests:

  ```
  {
      "Statement": [
          {
              "Effect": "Allow",
              "Action": "ots:*",
              "Resource": "acs:ots:*:*:*",
              "Condition": {
                  "Bool": {
                      "acs:SecureTransport": "true"
                  }
              }
          }
      ],
      "Version": "1"
  }
  ```

- MFA-based access control

  RAM allows you to specify whether requests that pass MFA can access Tablestore resources.

  The following sample code only allows requests that have passed MFA:

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "ots:*",
            "Resource": "acs:ots:*:*:*",
            "Condition": {
                "Bool": {
                    "acs:MFAPresent ": "true"
                }
            }
        }
    ],
    "Version": "1"
}
```

- Time-based access control

  RAM allows you to specify the access time of requests. Access requests earlier than the specified time are allowed or denied. The following example shows a typical application scenario.

  Example: RAM users can access resources only before 00:00:00 January 1, 2016 (UTC+8).

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "ots:*",
            "Resource": "acs:ots:*:*:*",
            "Condition": {
                "DateLessThan": {
                    "acs:CurrentTime": "2016-01-01T00:00:00+08:00"
                }
            }
        }
    ],
    "Version": "1"
}
```

Scenarios

This section describes the specific policies and authorization methods that are supported for the Action, Resource, and Condition parameters.

- Multiple authorization conditions

  In this scenario, RAM users that use the 10.101.168.111/24 CIDR block can manage the instances named online-01 and online-02 and all tables in these instances, including reading data from and writing data to the tables. Access is allowed only over HTTPS before 00:00:00 January 1, 2016.

  To configure multiple authorization conditions, perform the following steps:

  i. Log on to the RAM console. By default, RAM is activated.

  ii. In the left-side navigation pane, choose **Permissions > Policies**.

  iii. On the **Policies** page, click **Create Policy**.

iv. Configure the **Policy Name** parameter and select **Script**. Enter the following content in the **Policy Document** field:

```
{
"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"
                ]
            },
            "DateLessThan": {
                "acs:CurrentTime": "2016-01-01T00:00:00+08:00"
            },
            "Bool": {
                "acs:SecureTransport": "true"
            }
        }
    }
],
"Version": "1"
}
```

v. Click **OK**.

vi. In the left-side navigation pane, choose **Identities > Users**. On the Users page, find the RAM user that you want to manage and click **Add Permissions** in the Actions column.

vii. In the Add Permissions panel, search for the new policy, and add the policy to the Selected column. Click **OK**. The selected policy is attached to the RAM user.

- Reject requests

  In this scenario, RAM users that use the IP address 10.101.169.111 cannot write data to tables that belong to instances whose names contain the online or product prefix, and are located in the China (Beijing) region. This policy does not take effect for operations on instances.

  To reject requests, follow the preceding steps to create a custom permission policy and attach the policy to the RAM user. You need to copy the following content to the **Policy Document** field when you create the policy:

```
{
    "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"
}
```