

Alibaba Cloud Table Store **Developer Guide**

Issue: 20191127

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 due to product version upgrades, adjustments, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and the updated versions of this document will be occasionally released through Alibaba Cloud-authorized channels. You shall 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 the document in the context that Alibaba Cloud products and services are provided on an "as is", "with all faults" and "as available" basis. 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 bear any liability for any errors or financial losses incurred by any organizations, companies, or individuals arising from their download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, bear responsibility for any indirect, consequent

ial, exemplary, incidental, 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 contact Alibaba Cloud directly if you discover any errors in this document

.

Document conventions

Style	Description	Example
	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.
	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.
	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.
	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 <code>cd /d C:/window</code> command to enter the Windows system folder.
<i>Italic</i>	Italic formatting is used for parameters and variables.	<code>bae log list --instanceid</code> <code>Instance_ID</code>
[] or [a b]	This format is used for an optional value, where only one item can be selected.	<code>ipconfig [-all -t]</code>

Style	Description	Example
<code>{}</code> or <code>{a b}</code>	This format is used for a required value, where only one item can be selected.	<code>switch {active stand}</code>

Contents

Legal disclaimer.....	I
Document conventions.....	I
1 Overview.....	1
2 Limits.....	3
3 Features and regions.....	6
4 Terms.....	8
4.1 Instance.....	8
4.2 Endpoint.....	10
4.3 Read/write throughput.....	11
4.4 Region.....	14
5 Wide column model.....	16
5.1 Introduction.....	16
5.2 Primary keys and attributes.....	17
5.3 Data versions and time to live.....	18
5.4 Naming conventions and data types.....	21
5.5 Data operations.....	22
5.6 Auto-increment function of the primary key column.....	35
5.7 Conditional update.....	37
5.8 Atomic counters.....	40
6 Timeline model.....	44
6.1 Introduction.....	44
6.2 Quick start.....	45
6.3 Basic operations.....	46
6.3.1 Overview.....	46
6.3.2 Initialization.....	46
6.3.3 Meta management.....	48
6.3.4 Timeline management.....	50
6.3.5 Queue management.....	51
7 Search Index.....	54
7.1 Overview.....	54
7.2 Features.....	56
7.3 API operations.....	60
7.3.1 Overview.....	60
7.3.2 CreateSearchIndex.....	67
7.3.3 DescribeSearchIndex.....	70
7.3.4 ListSearchIndex.....	70
7.3.5 DeleteSearchIndex.....	71
7.3.6 Array and Nested field types.....	71

7.3.7 Sort.....	73
7.3.8 Tokenization.....	73
7.3.9 MatchAllQuery.....	77
7.3.10 MatchQuery.....	77
7.3.11 MatchPhraseQuery.....	79
7.3.12 TermQuery.....	80
7.3.13 TermsQuery.....	81
7.3.14 PrefixQuery.....	82
7.3.15 RangeQuery.....	82
7.3.16 WildcardQuery.....	83
7.3.17 BoolQuery.....	84
7.3.18 Nested query.....	86
7.3.19 GeoDistanceQuery.....	87
7.3.20 GeoBoundingBoxQuery.....	88
7.3.21 GeoPolygonQuery.....	89
7.3.22 ExistQuery.....	90
7.3.23 Statistics.....	91
7.3.24 Aggregation.....	98
7.4 Limits.....	106
8 Global secondary index.....	111
8.1 Overview.....	111
8.2 Introduction.....	113
8.3 Scenarios.....	115
8.4 Java SDK for global secondary indexes.....	124
8.5 APIs.....	128
8.6 Appendix.....	128
9 Tunnel service.....	130
9.1 Overview.....	130
9.2 Features.....	131
9.3 Description of the data consumption framework.....	132
9.4 Quick start.....	135
9.5 SDKs.....	137
9.6 Incremental synchronization performance white paper.....	137
10 HBase.....	145
10.1 Table Store HBase Client.....	145
10.2 Table Store HBase Client supported functions.....	146
10.3 Differences between Table Store and HBase.....	152
10.4 Migrate from HBase to Table Store.....	157
10.5 Migrate HBase of an earlier version.....	160
10.6 Hello World.....	162
11 Authorization management.....	167
11.1 RAM and STS.....	167
11.2 Create a RAM user account.....	170
11.3 Grant permissions to a RAM user.....	171

11.4 Configure an MFA device for a RAM user.....	172
11.5 STS temporary access authorization.....	173
11.5.1 Create a temporary role and grant permissions.....	173
11.5.2 Authorize temporary access.....	176
11.6 Custom permissions.....	181
11.7 Authorize a RAM user account to log on to the console.....	192
11.8 Examples.....	193

1 Overview

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

Terms

The following table describes the terms for Table Store.

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

Models

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

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

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

Features

The following table describes the features of Table Store.

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

2 Limits

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

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

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

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

3 Features and regions

Most Table Store features are supported in all regions. This topic describes the features that are not supported by all regions or are currently in invitational preview.

**Note:**

If the feature you are searching for is not included on this list, it means that this feature is supported in all regions.

Features in invitational preview and supported regions

Table Store has the following features in invitational preview:

- **Global secondary index:** supported in all regions. You must submit a ticket to enable this feature.

Features not supported in all regions

Search index and *Tunnel Service* are not supported in all regions. The following table describes which regions support the feature.

Region	Search index	Tunnel Service
China (Hangzhou)	Yes	Yes
Finance Cloud of China (Hangzhou)	-	Yes
China (Shanghai)	Yes	Yes
Finance Cloud of China (Shanghai)	-	Yes
China (Qingdao)	-	-
China (Beijing)	Yes	Yes
China (Zhangjiakou-Beijing Winter Olympics)	Yes	Yes
China (Hohhot)	-	-
China (Shenzhen)	Yes	Yes
Finance Cloud of China (Shenzhen)	-	-
China (Chengdu)	-	-

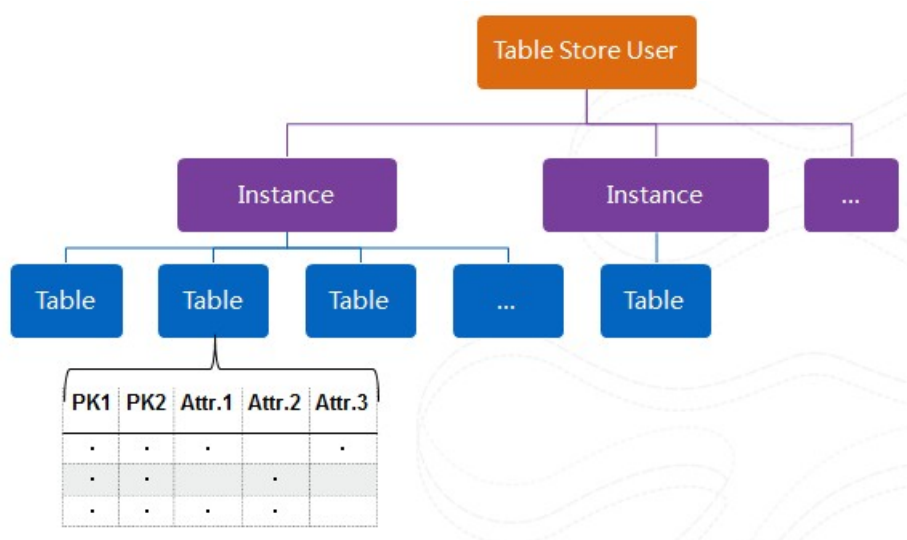
Region	Search index	Tunnel Service
China (Hong Kong)	Yes	Yes
Japan (Tokyo)	-	-
Singapore	Yes	Yes
Australia (Sydney)	-	-
Malaysia (Kuala Lumpur)	-	-
Indonesia (Jakarta)	-	-
UAE (Dubai)	-	-
US (Silicon Valley)	-	-
Germany (Frankfurt)	-	-
US (Virginia)	-	-
India (Mumbai)	Yes	Yes
UK (London)	-	-

4 Terms

4.1 Instance

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

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



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

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

Naming rule

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

- Contain English letters, numbers, and hyphens(-)
- Start with English letters

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

Instance type

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



Notice:

An instance type cannot be modified once the instance is created.

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

- **High-performance instance**

High-performance instances support millions of read-write transactions per second (TPS) with 1 ms average latency of read and write operations per row.

High-performance instances are suitable for scenarios requiring high read and write performance and concurrency, such as gaming, financial risk control, social networking applications, product recommendation systems, and public opinion sensing.

- **Capacity instance**

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



Notice:

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

Instance type supported by region

Region Name	High-performance instance	Capacity instance
China East 1 (Hangzhou)	Supported	Supported
China East 2 (Shanghai)	Supported	Supported
China North 2 (Beijing)	Supported	Supported
China North 3 (Zhangjiakou)	In development	Supported
China North 5 (Huhehaote)	In development	Supported
China South 1 (Shenzhen)	Supported	Supported
China(Hong Kong)	In development	Supported
Singapore	Supported	In development
US East 1 (Virginia)	Supported	In development
US West 1 (Silicon Valley)	Supported	In development
Asia Pacific NE 1 (Japan)	In development	Supported
Germany 1 (Frankfurt)	In development	Supported
Middle East 1 (Dubai)	In development	Supported
Asia Pacific SE 2 (Sydney)	In development	Supported
Asia Pacific SE 3 (Kuala Lumpur)	In development	Supported
Asia Pacific SE 5 (Jakarta)	In development	Supported
Asia Pacific SOU 1 (Mumbai)	In development	Supported

4.2 Endpoint

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

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

```
https://instanceName.region.ots.aliyuncs.com
```

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

```
https://instanceName.region.ots-internal.aliyuncs.com
```

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

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

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

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

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

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

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

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

4.3 Read/write throughput

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

- One read CU indicates that 4 KB data is read from the table.
- One write CU indicates that 4 KB data is written into the table.

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

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

Reserved throughput

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

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

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

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



Note:

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

- The reserved read/write throughput of a non-existent table is regarded as zero.
- To access a non-existent table, one additional read CU or one additional write CU is consumed depending on the actual operation.

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

Additional throughput

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

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

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

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



Note:

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

For the additional read/write throughput mode, it is difficult to estimate the amount of compute resources that need to be reserved for data tables. Table Store is required to provide sufficient service capability to effectively handle access traffic spikes. For this reason, the unit price of additional read/write throughput is

higher than that of reserved read/write throughput. To make sure that low costs are maintained, we recommend that you set an appropriate value of the reserved read/write throughput.

**Note:**

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

4.4 Region

Region refers to a service region of Alibaba Cloud.

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

The following table lists the regions supported by Table Store.

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

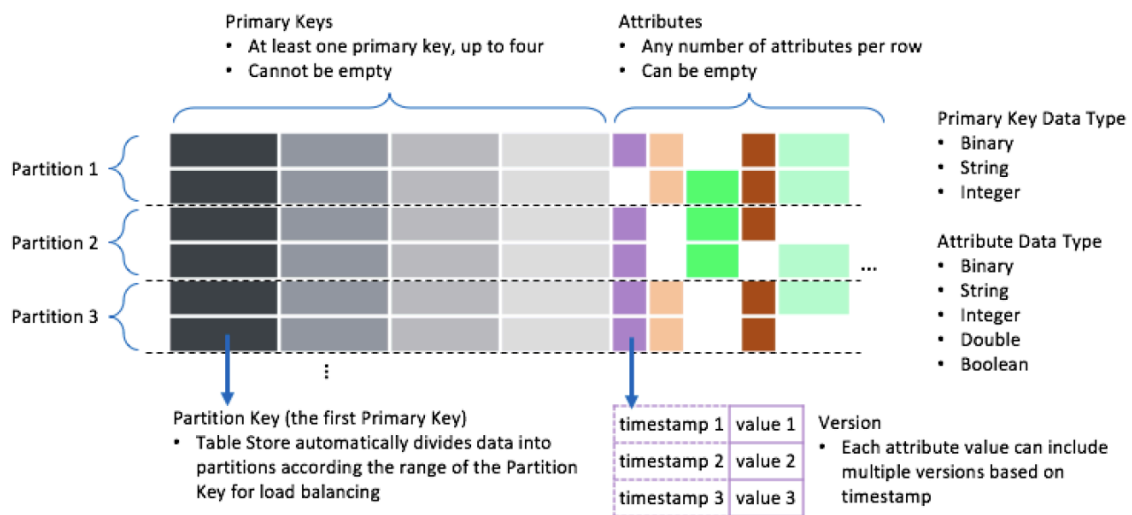
Region Name	RegionID
Asia Pacific SE 5 (Jakarta)	ap-southeast-5
Asia Pacific SOU 1 (Mumbai)	ap-south-1

5 Wide column model

5.1 Introduction

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

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



The Wide Column model consists of the following parts:

- **Primary key:** Every row has a primary key with a multi-column structure (1-4 columns). The primary key is defined as a fixed schema, and is used primarily to uniquely distinguish a row of data.
- **Partition key:** The first column of the primary key is called a partition key. The partition key is used to partition the table by range. Every partition is distributively dispatched to services on different machines. Within the same partition key, we provide cross-row transactions. For more information, see [Primary key and attribute](#).
- **Attribute column:** In one row, with the exception of the primary key, all other columns are attribute columns. Attribute columns correspond to many values. Different values correspond to different versions, and each row stores an unlimited number of attribute columns.

- **Version:** Each value corresponds to a different version that acts as a timestamp to define the time to live of that data.
- **Data type:** Table Store allows many different data types, including String, Binary, Double, Integer and Boolean.
- **Time To Live (TTL):** Each table defines the amount of time a data can be stored before being deleted. For example, if the TTL is defined as one month, the data written into the table more than a month ago will be cleared automatically. The write time of the data is determined by the version number. This write time is usually taken from the server time, but it can also be determined by the time specified by the application. For more information, see [Data versions and Time To Live](#).
- **Max versions:** Each table defines the maximum number of version data that can be stored in a column, which is used to control the number of versions in each column. If the number of versions in an attribute column exceeds the value in max versions, the earliest version is deleted.

5.2 Primary keys and attributes

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

Primary keys

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

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

Partition keys

The first column of a primary key is called the partition key. Table store assigns a row of data to the corresponding partitions determined by the range of each row's partition keys to achieve load balancing. Rows that have the same partition key value belong to the same partition. A partition may store rows with multiple

partition key values. Table Store separates a partition or merges multiple partitions based on specific rules. This process is completed automatically.

**Note:**

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

Attributes

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

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

5.3 Data versions and time to live

Version numbers

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

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

- Time to live (TTL)

The version number can be used to determine the lifecycle of a table. Assume that a version number of an attribute is 1468944000000, which is calculated based on the time of July 20, 2016, 00:00:00 UTC. When you set the TTL as 86400

(one day), the data of that version expires on July 21, 2016, 00:00:00 UTC. Then, the data is automatically deleted.

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

- Read the version number of each row's data

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

Max Versions

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

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



Note:

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

Max Version Offset

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

To ensure that data is written successfully, Table Store will check the version number of an attribute when processing write requests. The range of valid version numbers of an attribute is: [The time when you write data - Max Version Offset, The time when you write data + Max Version Offset). The version number of an attribute is measured in milliseconds. After the version number is divided by 1000, the result that is measured in seconds must fall within this range. When a version number does not fall within the range, this write request fails.

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

Time to live

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

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



Note:

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

5.4 Naming conventions and data types

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

Naming conventions

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

Item	Description
Structure	A name can contain uppercase letters (A to Z), lowercase letters (a to z), digits (0 to 9), and underscores (_).
First character	A name must start with an uppercase letter (A to Z), a lowercase letter (a to z), or an underscore (_).
Case sensitivity	A name is case-sensitive.
Length	A name can be 1 to 255 characters in length.
Uniqueness	<ul style="list-style-type: none">A table name must be unique under the same instance.Table names under different Table Store instances can be the same.

Data types of primary key columns

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

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

Data types of attribute columns

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

Data type	Description	Size limit
String	Data is in UTF-8. Empty strings are allowed.	For more information, see Limits .
Integer	Data is 64-bit long.	Up to 8 Bytes
Double	Data is 64-bit long.	Up to 8 Bytes

Data type	Description	Size limit
Boolean	The value can be True or False.	Up to 1 Byte
Binary	Data is binary. Empty values are allowed.	For more information, see Limits .

5.5 Data operations

In Table Store, tables are composed of rows. Each row includes primary keys and attributes. This topic introduces data operations of Table Store.

Table Store rows

A row is a basic feature in the creation of tables in Alibaba Cloud Table Store. Rows are composed of primary keys and attributes. A primary key is required and all rows in the same table must have the same primary key column name and type. Attributes are not required and each row may have different attributes.

Table Store data operations include three types:

- **Single row operations**
 - **GetRow:** Read a single row from the table.
 - **PutRow:** Insert a row into the table. If the row already exists, the existing row is deleted and the new row is written.
 - **UpdateRow:** Update a row from the table. You can add or delete attribute columns of an existing row or update the value of an existing attribute column. If the row does not exist, this operation adds a new row.
 - **DeleteRow:** Delete a row from the table.
- **Batch operations**
 - **BatchGetRow:** Batch read data from multiple rows with one request.
 - **BatchWriteRow:** Batch insert, update, or delete multiple rows in one request.
- **Range read operations**
 - **GetRange:** Read data from a table within a certain range.

Single row operations

Single row write operations

- Table Store has three single row write operations. The following are the descriptions and considerations of these operations.
 - **PutRow:** Write a new row. If this row already exists, the existing row is deleted and the new row is written.
 - **UpdateRow:** Update the data of a row. Based on the request content, Table Store adds new columns or modifies/deletes the specified column values for this row. If this row does not exist, a new row is inserted. However, an UpdateRow request with only deletion instructions onto a row that does not exist does not insert a new row.
 - **DeleteRow:** Delete a row. If the row to be deleted does not exist, nothing happens.

By setting the condition field in the request, you can specify whether a row existence check is performed before executing the write operation. Three condition checking options are available.

- **IGNORE:** The row existence check is not performed.
- **EXPECT_EXIST:** The row is expected to exist. The operation succeeds only if the row exists. Otherwise, the operation fails.
- **EXPECT_NOT_EXIST:** The row is not expected to exist. The operation succeeds only if the row does not exist. Otherwise, the operation fails.

If the condition checking is **EXPECT_NOT_EXIST**, DeleteRow and UpdateRow fail because it is meaningless to delete or update the non-existing rows. If the condition checking is **EXPECT_EXIST**, you can use PutRow to update a non-existing row.

If the operation fails, such as a parameter check failure, the data size of the row is too large, or the existence check fails, an error code is returned to the application. Applications receive the number of consumed capacity units (CU) for successful operations.

The rules for calculating the number of write capacity units (CU) consumed in each operation are defined as follows.

- **PutRow:** The sum of the data size of the primary key of the modified row and the data size of the attribute column is divided by 4 KB and rounded up. If the row existence check condition is not **IGNORE**, a number of read CUs are consumed. This is equivalent to the rounded up value after dividing the data

size of the primary key of this row by 4 KB. If an operation does not meet the row existence check condition specified by the application, the operation fails and consumes 1 write CU and 1 read CU. For more information, see [API Reference - PutRow](#).

- **UpdateRow:** The sum of the data size of the primary key of the modified row and the data size of the attribute column is divided by 4 KB and rounded up. If UpdateRow contains an attribute column which must be deleted, only the column name is calculated into the data size of this attribute column. If the row existence check condition is not IGNORE, a number of read CUs are consumed. This is equivalent to the rounded up value after dividing the data size of the primary key of this row by 4 KB. If an operation does not meet the row existence check condition specified by the application, the operation fails and consumes 1 write CU and 1 read CU. For more information, see [API Reference - UpdateRow](#).
- **DeleteRow:** The data size of the primary key of the deleted row is divided by 4 KB and rounded up. If the row existence check condition is not IGNORE, a number of read CUs are consumed. This is equivalent to the rounded up value after dividing the data size of the primary key of this row by 4 KB. If an operation does not meet the row existence check condition specified by the application, the operation fails and consumes 1 write capacity unit. For more information, see [API Reference - DeleteRow](#).

Also, a certain number of read CUs are consumed by write operations based on specified conditions.

Examples

The following examples illustrate how the number of write CUs are calculated for single row write operations.

Example 1: Use PutRow to write a row.

```
//PutRow operation
//row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+
3000Byte=4322Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte), 'value2':String(3000Byte)
}
}

//Original row
//row_size=len('pk')+len('value2')+8Byte+900Byte=916Byte
```

```
//row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
    attributes:{'value2':String(900Byte)}
}
```

The consumption of the read/write CUs for PutRow is described as follows.

- When the existence check condition is set to EXPECT_EXIST: The number of write CUs consumed is the rounded up value after dividing 4,322 bytes by 4 KB, and the number of read CUs is the rounded up value after dividing 10 bytes (data size of the primary key of the row) by 4 KB. Therefore, PutRow consumes 2 write CUs and 1 read CU.
- When the existence check condition is set to IGNORE: The number of write CUs consumed is the rounded up value after dividing 4,322 bytes by 4 KB. No read CU is consumed. Therefore, PutRow consumes 1 write CU and 0 read CU.
- When the existence check condition is set to EXPECT_NOT_EXIST: The existence check condition of the specified row fails. PutRow consumes 1 write CU and 1 read CU.

Example 2: Use UpdateRow to write a new row.

```
//UpdateRow operation
//Length of attribute column deleted is calculated for row size
//row_size=len('pk')+len('value1')+len('value2')+8Byte+900Byte=
922Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte), 'value2':Delete}
}

//The original row does not exist
//row_size=0
```

The consumption of read/write CUs for UpdateRow is described as follows.

- When the existence check condition is set to IGNORE: The number of write CUs consumed is the rounded up value after dividing 922 bytes by 4 KB. No

read CU is consumed. Therefore, UpdateRow consumes 1 write CU and 0 read CU.

- When the existence check condition is set to EXPECT_EXIST: The existence check condition of the specified row fails. UpdateRow consumes 1 write CU and 1 read CU.

Example 3: Use UpdateRow to update an existing row.

```
//UpdateRow operation
//row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+
3000Byte=4322Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte), 'value2':String(3000Byte
)}
}
//Original row
//row_size=len('pk')+len('value1')+8Byte+900Byte=916Byte
//row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte)}
}
```

The consumption of read/write CUs for UpdateRow is described as follows.

- When the existence check condition is set to EXPECT_EXIST: The number of write CUs consumed is the rounded up value after dividing 4,322 bytes by 4 KB, and the number of read CUs is the rounded up value after dividing 10 bytes (data size of the primary key of the row) by 4 KB. Therefore, UpdateRow consumes 2 write CUs and 1 read CU.
- When the existence check condition is set to IGNORE: The number of write CUs consumed is the rounded up value after dividing 4,322 bytes by 4 KB. No read CU is consumed. Therefore, UpdateRow consumes 1 write CU and 0 read CU.

Example 4: Use DeleteRow to delete a non-existent row.

```
//The original row does not exist
//row_size=0

//DeleteRow operation
//row_size=0
//row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
}
```

The data size before and after modification is 0. Whether the DeleteRow operation succeeds or fails, at least 1 write CU is consumed. Therefore, this DeleteRow consumes 1 write CU.

The consumption of read/write CUs for DeleteRow is described as follows.

- When the existence check condition is set to EXPECT_EXIST: The number of write CUs consumed is the rounded up value after dividing 10 bytes (data size of the primary key of the row) by 4 KB. No read CU is consumed. Therefore, DeleteRow consumes 1 write CU and 0 read CU.
- When the existence check condition is set to IGNORE: The number of write CUs consumed is the rounded up value after dividing 10 bytes (data size of the primary key of the row) by 4 KB, and the number of read CUs is the rounded up value after dividing 10 bytes by 4 KB. Therefore, DeleteRow consumes 1 write CU and 1 read CU.

For more information, see [PutRow](#), [UpdateRow](#), and [DeleteRow](#).

Single row read operations

- GetRow is the only single row read operation.

Applications provide the complete primary key and names of all columns to be returned. The column names can be either the primary key or attribute columns. Users are also able to not specify any column names to be returned, in which case all row data is returned.

Table Store calculates the consumed read CUs by adding the data size of the primary key of the read row and the data size of the read attribute column. The data size is divided by 4 KB and rounded up as the number of read CUs consumed in this read operation. If the specified row does not exist, 1 read CU is consumed. Single row read operations do not consume write CUs.

Example

This example illustrates how the number of read CUs consumed by GetRow is calculated.

```
//GetRow operation  
  
//row_size=len('pk')+len('value1')+len('value2')+8Byte+1200Byte+  
3100Byte=4322Byte  
{
```

```
    primary_keys: {'pk': 1},
    attributes: {'value1': String(1200Byte), 'value2': String(3100Byte)
  }}
}

//GetRow operation
//Reading data size=len('pk')+len('value1')+8Byte+1200Byte=1216Byte
{
  primary_keys: {'pk': 1},
  columns_to_get: {'value1'}
}
```

The number of consumed read CUs is rounded up after dividing 1218 Bytes by 4KB. This GetRow consumes 1 read CU.

For more information, see [GetRow](#).

Multi-Row operations

Table Store provides two multi-row operations: BatchWriteRow and BatchGetRow.

- **BatchWriteRow:** Used to insert, modify, or delete multiple rows from one or more tables. BatchWriteRow can be considered as a batch of multiple PutRow, UpdateRow, and DeleteRow operations. The sub-operations in a single BatchWriteRow are executed independently. Table Store separately returns the execution results for each sub-operation to the application. Sometimes, parts of the request are successful, while other parts fail. Even if an error is not returned for the overall request, the application must still check the return results for each sub-operation to determine the actual status. The write CUs consumed by each BatchWriteRow sub-operation are calculated independently.
- **BatchGetRow:** Used to read multiple rows from one or more tables. In BatchGetRow, each sub-operation is executed independently. Table Store separately returns the execution results for each sub-operation to the application. Sometimes, parts of the request are successful, while other parts fail. Even if an error is not returned for the overall request, the application must still check the return results for each sub-operation to determine the actual status. The read capacity units consumed by each BatchGetRow sub-operation are calculated independently.

For more information, see [BatchWriteRow](#) and [BatchGetRow](#).

Range read operations

GetRange is a range read operation in Table Store. GetRange returns data in a specified range of primary keys to applications.

Rows in Table Store tables are sorted in ascending order of primary keys. `GetRange` specifies a left-closed-right-open range, and returns data from rows with primary keys in this range. End points of ranges are composed of either effective primary keys or the virtual points: `INF_MIN` and `INF_MAX`. The number of columns for the virtual point must be the same as that of the primary key. Here, `INF_MIN` represents an infinitely small value, so any values of other types are always greater. `INF_MAX` represents an infinitely large value, so any values of other types are always less.

`GetRange` must specify the columns to be retrieved by their names. The request column name can contain multiple column names. If a row has a primary key in the read range, but does not contain another column specified for return, the results returned by the request do not contain data from this row. If columns to be retrieved are not specified, the entire row is returned.

`GetRange` must specify the read direction, which can be either forward or backward. For example, a table has two primary key columns A and B, and $A < B$. When `[A, B)` is read in the forward direction, rows with the primary key greater than or equal to A and less than B are returned in the order A to B. When `[B,A)` is read in the backward direction, rows greater than A and less than or equal to B are returned in the order B to A.

`GetRange` can specify the maximum number of returned rows. Table Store ends the operation as soon as the maximum number of rows are returned according to the forward or backward direction, even if some rows remain unreturned in the specified ranges.

`GetRange` may stop execution and return data to the application in the following situations:

- The total size of row data to be returned reaches 4 MB.
- The number of rows to be returned is equal to 5000.
- The number of returned rows is equal to the maximum number of rows specified in requests to be returned.
- In premature-return situations, responses returned by `GetRange` contain the primary key for the next row of unread data. Applications can use this value as the starting point for subsequent `GetRange` operations. If the primary key for

the next unread row is null, this indicates all data in the read range has been returned.

Table Store accumulates the total data size of the primary key and attribute column read for all rows from the read range start point to the next row of unread data. The data size is then divided by 4 KB and rounded up to find the number of consumed read CUs. For example, if the read range contains 10 rows and the primary key and actual data size of the attribute column read for each row is 330 Bytes, the number of consumed read CU is 1 (divide the total read data size 3.3 KB by 4 KB and rounded up to 1).

Examples

The following examples illustrate how the number of read CUs are calculated for GetRange. In these examples, the table contents are as follows. PK1 and PK2 are the table's primary key columns, and their types are String and Integer, respectively. A and B are the table's attribute columns.

PK1	PK2	Attr1	Attr2
'A'	2	'Hell'	'Bell'
'A'	5	'Hello'	Non-exist
'A'	6	Non-exist	'Blood'
'B'	10	'Apple'	Non-exist
'C'	1	Non-exist	Non-exist
'C'	9	'Alpha'	Non-exist

Example 1: Read Data in a specified range.

```
//Request
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 2)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INTEGER, 1)

//Response
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
    attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
    "Bell")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
    attribute_columns:("Attr1", STRING, "Hello")
  },
}
```



```

    {
      primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
      attribute_columns:("Attr2", STRING, "Blood")
    },
    {
      primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
      attribute_columns:("Attr1", STRING, "Apple")
    }
  }
}

```

Example 2: Use INF_MIN and INF_MAX to read all data in a table.

```

// Request
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", INF_MIN)
exclusive_end_primary_key: ("PK1", INF_MAX)

//Response
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
    attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
"Bell")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
    attribute_columns:("Attr1", STRING, "Hello")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
    attribute_columns:("Attr2", STRING, "Blood")
  },
  {
    primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
    attribute_columns:("Attr1", STRING, "Apple")
  }
  {
    primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 1)
  }
  {
    primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 9)
    attribute_columns:("Attr1", STRING, "Alpha")
  }
}
}

```

Example 3: Use INF_MIN and INF_MAX in certain Primary Key columns.

```

// Request
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)

//Response
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)

```

```

        attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
"Bell")
    },
    {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
        attribute_columns:("Attr1", STRING, "Hello")
    },
    {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
        attribute_columns:("Attr2", STRING, "Blood")
    }
}

```

Example 4: Backward reading.

```

// Request
table_name: "table_name"
direction: BACKWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INTEGER, 1)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 5)

//Response
cosumed_read_capacity_unit: 1
rows: {
    {
        primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 1)
    },
    {
        primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
        attribute_columns:("Attr1", STRING, "Apple")
    },
    {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
        attribute_columns:("Attr2", STRING, "Blood")
    }
}

```

Example 5: Specify a column name not including a PK.

```

// Request
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MAX)
columns_to_get: "Attr1"

//Response
cosumed_read_capacity_unit: 1
rows: {
    {
        attribute_columns: {"Attr1", STRING, "Alpha"}
    }
}

```

Example 6: Specify a column name including a PK.

```

// Request
table_name: "table_name"

```

```

direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MAX)
columns_to_get: "Attr1", "PK1"

//Response
consumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "C")
  }
  {
    primary_key_columns:("PK1", STRING, "C")
    attribute_columns:("Attr1", STRING, "Alpha")
  }
}

```

Example 7: Use limit and breakpoints.

```

//Request 1
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)
limit: 2

//Response 1
consumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
    attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
"Bell")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
    attribute_columns:("Attr1", STRING, "Hello")
  }
}
next_start_primary_key:("PK1", STRING, "A"), ("PK2", INTEGER, 6)

// Request 2
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 6)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)
limit: 2

// Response 2
consumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
    attribute_columns:("Attr2", STRING, "Blood")
  }
}

```

Example 8: Use GetRange to calculate the consumed read CUs.

GetRange is performed on the following table. PK1 is the table's primary key column, Attr1 and Attr2 are the attribute columns.

PK1	Attr1	Attr2
1	Non-existent	String(1000Byte)
2	8	String(1000Byte)
3	String(1000Byte)	Non-existent
4	String(1000Byte)	String(1000Byte)

```
// Request
table_name: "table2_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", INTEGER, 1)
exclusive_end_primary_key: ("PK1", INTEGER, 4)
columns_to_get: "PK1", "Attr1"

//Response
consumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", INTEGER, 1)
  },
  {
    primary_key_columns:("PK1", INTEGER, 2),
    attribute_columns:("Attr1", INTEGER, 8)
  },
  {
    primary_key_columns:("PK1", INTEGER, 3),
    attribute_columns:("Attr1", STRING, String(1000Byte))
  },
}
```

For this GetRange request:

- Data size of the first row: $\text{len}(\text{'PK1'}) + 8 \text{ Bytes} = 11 \text{ Bytes}$
- Data size of the second row: $\text{len}(\text{'PK1'}) + 8 \text{ Bytes} + \text{len}(\text{'Attr1'}) + 8 \text{ Bytes} = 24 \text{ Bytes}$
- Data size of the third row: $\text{len}(\text{'PK1'}) + 8 \text{ Bytes} + \text{len}(\text{'Attr1'}) + 1000 \text{ Bytes} = 1016 \text{ Bytes}$

The number of consumed read CUs is the rounded up value after dividing 1051 Bytes (11 Bytes + 24 Bytes + 1016 Bytes) by 4 KB. So this GetRange consumes 1 read CU.

For more information, see [GetRange](#).

Best Practice

Data operations

Table Store SDK for data operation

Use Table Store Java SDK for table operations

Use Table Store Python SDK for table operations

5.6 Auto-increment function of the primary key column

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

Features

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

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

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

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

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

Limits

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

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

Interface

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

You cannot change the auto-increment attribute of a table by using UpdateTable.
- PutRow/UpdateRow/BatchWriteRow
 - When writing the table, you do not need to set specific values for the column that you want to set as auto-incrementing. You only need to set a placeholder,

for example, `AUTO_INCREMENT`. For more information, see [Primary key column auto-increment](#).

- You can set `ReturnType` in `ReturnContent` as `RT_PK`, that is, to return the complete primary key value, which can be used in the `GetRow` query.
- `GetRow/BatchGetRow`

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

- Other interfaces

Not changed

Usage

Java SDK: Auto-increment of the primary key column

Billing

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

5.7 Conditional update

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

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

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

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



Note:

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

Procedure

1. Construct `SingleColumnValueCondition`.

```
// set condition Col0==0.
SingleColumnValueCondition singleColumnValueCondition = new
SingleColumnValueCondition("Col0",
    SingleColumnValueCondition.CompareOperator.EQUAL,
    ColumnValue.fromLong(0));
// If column Col0 does not exist, the condition check fails.
singleColumnValueCondition.setPassIfMissing(false);
// Only check the latest version
singleColumnValueCondition.setLatestVersionsOnly(true);
```

2. Construct `CompositeColumnValueCondition`.

```
// condition composite1 is (Col0 == 0) AND (Col1 > 100)
CompositeColumnValueCondition composite1 = new CompositeColumnValue
Condition(CompositeColumnValueCondition.LogicOperator.AND);
SingleColumnValueCondition single1 = new SingleColumnValueCondition
("Col0",
    SingleColumnValueCondition.CompareOperator.EQUAL,
    ColumnValue.fromLong(0));
SingleColumnValueCondition single2 = new SingleColumnValueCondition
("Col1",
    SingleColumnValueCondition.CompareOperator.GREATER_THAN,
    ColumnValue.fromLong(100));
composite1.addCondition(single1);
composite1.addCondition(single2);

// condition composite2 is ( (Col0 == 0) AND (Col1 > 100) ) OR (
Col2 <= 10)
CompositeColumnValueCondition composite2 = new CompositeColumnValue
Condition(CompositeColumnValueCondition.LogicOperator.OR);
SingleColumnValueCondition single3 = new SingleColumnValueCondition
("Col2",
    SingleColumnValueCondition.CompareOperator.LESS_EQUAL,
    ColumnValue.fromLong(10));
composite2.addCondition(composite1);
```



```
composite2.addCondition(single3);
```

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

```
private static void updateRowWithCondition(SyncClient client,
String pkValue) {
    // construct the primary
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(pkValue));
    PrimaryKey primaryKey = primaryKeyBuilder.build();

    // read a row
    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
();

    // Col0 = Col0 + 1 by conditional update
    RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);
    Condition condition = new Condition(RowExistenceExpectation.
EXPECT_EXIST);
    ColumnCondition columnCondition = new SingleColumnValueCon
dition("Col0", SingleColumnValueCondition.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());
    }
}
```

Example

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

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

```
Update(new_value);
```

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

Billing

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

5.8 Atomic counters

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

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

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

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

Methods

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

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

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

```
private static void incrementByUpdateRowApi(SyncClient client) {
    // You can specify a primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString("pk0"));
    PrimaryKey primaryKey = primaryKeyBuilder.build();

    RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);

    // You can increment the price value by 10 without specifying
a timestamp.
    rowUpdateChange.increment(new Column("price", ColumnValue.
fromLong(10)));

    // You can specify a flag to indicate that the updated value
of the atomic counter must be returned.
    rowUpdateChange.addReturnColumn("price");
    rowUpdateChange.setReturnType(ReturnType.RT_AFTER_MODIFY);

    // You can update the price attribute.
    UpdateRowResponse response = client.updateRow(new UpdateRowR
equest(rowUpdateChange));

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



Note:

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

Scenarios

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

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

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

Restrictions

Note the following restrictions when using atomic counters:

- Atomic counters only support the Integer type.
- The default value of an empty atomic counter is zero. When you implement an atomic counter on an existing attribute with a non-Integer type, an OTSParameterInvalid error occurs.
- You can update an atomic counter by using a positive or a negative number, but you must avoid an integer overflow. If an overflow issue appears, an OTSParameterInvalid error occurs.
- When you modify an atomic counter, the value will not be returned by default. You can use addReturnColumn() and setReturnType() to specify the name and updated value of an atomic counter that will be returned.
- You cannot update an attribute and an atomic counter simultaneously for a single request. If you have incremented or decremented the attribute A, then you cannot perform other operations, such as overwrite and delete operations on the attribute A.
- You can perform multiple update operations on the same row using a BatchWriteRow request. When you perform an atomic counter operation on a row, other operations in this BatchWriteRow request cannot be performed on this row.

- You can only implement an atomic counter on an attribute with the latest version . After you perform the update operation on the atomic counter, the atomic counter will be specified with a new version.
- An error may occur when an atomic counter encounters network timeouts or system failures. You can retry the operation. An atomic counter may be updated twice. This symptom leads to an overcounting or undercounting issue. In this case, we recommend that you can use *conditional update* to precisely update the attribute.

6 Timeline model

6.1 Introduction

Overview

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

Architecture

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

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

Features

The Timeline model supports the following features:

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

Timeline

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

Table Store Java SDK (integrated with the Timeline model)

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

6.2 Quick start

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

Procedure

1. **Log on to the Table Store console and create a Table Store instance. For more information, see [#unique_41](#).**
2. **Download and install the Table Store Java SDK. For more information, see [#unique_42](#).**
3. **Determine an endpoint and configure an AccessKey pair to initialize the instance. For more information, see [#unique_43](#).**
4. **Download the [sample code](#) to get started with the Timeline model.**

6.3 Basic operations

6.3.1 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](#)

6.3.2 Initialization

Initialize the TimelineStore Factory

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

```
/**
 * Set the retry policy.
 * Code: configuration.setRetryStrategy(new DefaultRetryStrategy());
 */
ClientConfiguration configuration = new ClientConfiguration();

SyncClient client = new SyncClient(
    "http://instanceName.cn-shanghai.ots.aliyuncs.com",
    "accessKeyId",
    "accessKeySecret",
    "instanceName", configuration);

TimelineStoreFactory factory = new TimelineStoreFactoryImpl(client);
```

Initialize MetaStore

Create a schema for a Meta table. The schema includes parameters such as Identifier and MetaIndex. Create a Store that manages Meta data by using the

TimelineStore Factory. You need to specify the following parameters: Meta table name, index, table name, primary key field, index name, and index type.

```
TimelineIdentifierSchema idSchema = new TimelineIdentifierSchema.  
Builder()  
    .addStringField("timeline_id").build();  
  
IndexSchema metaIndex = new IndexSchema();  
metaIndex.addFieldSchema( //Configure the index field and index type.  
    new FieldSchema("group_name", FieldType.TEXT).setIndex(true).  
    setAnalyzer(FieldSchema.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 by using the parameters in metaSchema. Afterward, create and configure an index.

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

Delete a table

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

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

Initialize TimelineStore

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

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

```
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.Analyzer.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 DefaultTableStoreWriter to 5.  
    .withIndex("metaIndex", timelineIndex); //Set the index.  
  
TimelineStore timelineStore = serviceFactory.createTimelineStore(  
timelineSchema);
```

Create a table

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

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

Delete a table

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

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

6.3.3 Meta management

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

Insert

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

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

```
TimelineMeta meta = new TimelineMeta(identifier)
    .setField("fileName", "fieldValue");

timelineMetaStore.insert(meta);
```

Read

You can call this operation to read TimelineMeta data in one row based on the Identifier 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("fileName", "new value");

timelineMetaStore.update(meta);
```

Delete

You can call this operation to delete the TimelineMeta data in one row based on the Identifier 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 the native SDK class 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);
```

6.3.4 Timeline management

You can call the operations for the fuzzy query and bool query to manage Timeline data. The query operations work on the basis of the Search Index feature. Only the TimelineStore that has IndexSchema configured supports the query operations.

An index can be LONG, DOUBLE, BOOLEAN, KEYWORD, GEO_POINT, or TEXT type. The index attributes include Index, Store, Array, and Analyzer, and have the same descriptions as those of the Search Index feature. For more information, see [Overview](#).

Search

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

```
/**
 * Search timeline by 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 the SDK of Table Store. You can call the flush operation to trigger the process of

sending the undelivered messages in the Buffer to Table Store and wait until Table Store stores all these messages.

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

6.3.5 Queue management

Obtain a Queue instance

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

```
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 some operations, such as Store, StoreAsync, BatchStore, Delete, Update, UpdateAsync, Get, and Scan.

Store

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

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

StoreAsync

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

```
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);//Specify the
SequenceId value by manual.
```

BatchStore

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

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

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

Get

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

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

GetLatestTimelineEntry

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

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

GetLatestSequenceId

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

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

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

timelineQueue.scan(scanParameter);
```

7 Search Index

7.1 Overview

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

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

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

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

Index differences

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

Index type	Description	Scenario
Table	A table is similar to a big map . Tables only support queries based on primary key columns.	<ul style="list-style-type: none"> • You can specify the complete primary key columns. • You can specify the prefixes of primary key columns.
Global secondary index	You can create one or more global secondary indexes and issue query requests against these indexes. This way, you can perform queries based on the primary key columns of these indexes.	<ul style="list-style-type: none"> • You can determine the required columns in advance , and only a few columns are required. • You can specify the complete primary key columns or the prefixes of primary key columns.
Search index	Search index uses inverted indexes, Bkd-trees, and column-oriented storage to meet various query scenarios.	All query and analysis scenarios that the table and the global secondary index do not support .

Precautions

Index synchronization

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

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

TTL

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

max versions

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

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

Features

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

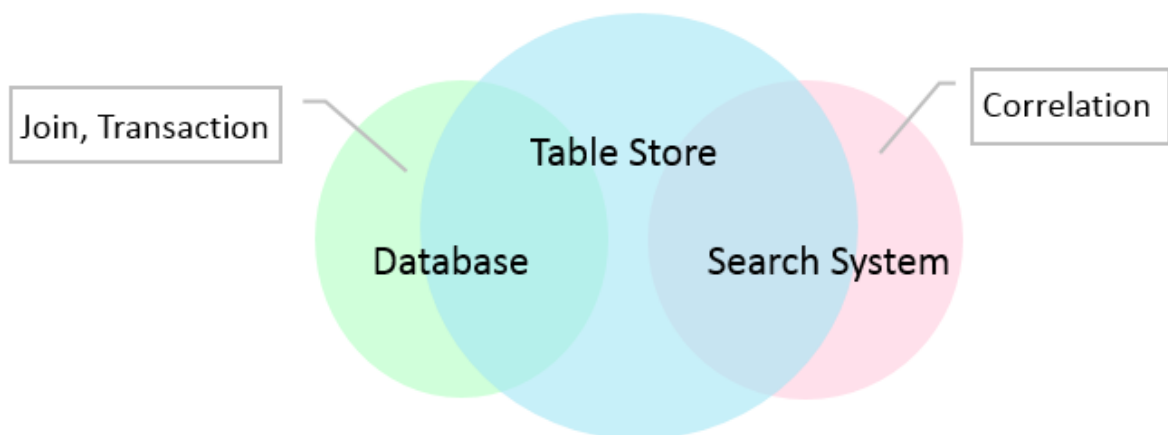


Table Store can provide all features of databases and search engines, except for join operations, transactions, and relevance of search results. Table Store also has high data reliability of databases and supports advanced queries of search engines. Therefore, Table Store can replace the common database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

7.2 Features

This topic describes the core features of search index.

Core features

Query based on non-primary key columns

Originally, Table Store only supports queries based on complete primary key columns or their prefixes. Queries based on non-primary key columns were not available in some scenarios. Search index enables Table Store to support queries

based on non-primary key columns. You only need to create a search index for the column to be queried.

Bool query

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

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

Query by geographical location

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

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

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

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

Full-text search

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

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

[Tokenization](#).

Fuzzy query

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

Prefix query

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

Nested query

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

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

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

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

Deduplication

Search index supports deduplication for query results. Deduplication allows you to specify the highest frequency of occurrence of an attribute value to achieve high cardinality. For example, when you search for a laptop on an e-commerce platform

, the first page may display products of a certain brand. This is not a user-friendly result. However, the deduplication feature of Table Store can resolve this issue.

Sorting

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

Total number of rows

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

SQL

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

SQL	Search index	Supported
SHOW	API operation: DescribeSearchIndex	Yes
SELECT	Parameter: ColumnsToGet	Yes
FROM	Parameter: index name	Supported for single indexes and not supported for multiple indexes
WHERE	Query: a variety of queries such as TermQuery	Yes
ORDER BY	Parameter: sort	Yes
LIMIT	Parameter: limit	Yes
DELETE	API operation: query followed by DeleteRow	Yes

SQL	Search index	Supported
LIKE	Query: wildcard query	Yes
AND	Parameter: operator = and	Yes
OR	Parameter: operator = or	Yes
NOT	Query: bool query	Yes
BETWEEN	Query: range query	Yes
NULL	ExistQuery	Yes

7.3 API operations

7.3.1 Overview

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

SDKs

You can use the following SDKs to implement search index.

- [Java SDK](#)
- [Python SDK](#)
- [Go SDK](#)
- [Node.js SDK](#)
- [.NET SDK](#)

API operations

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

Fields

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

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

**Notice:**

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

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

Attribute	Type	Option	Description
Index	Boolean	Specifies whether to create an index for a column.	<ul style="list-style-type: none"> • True indicates that Table Store creates an inverted index or spatial index for the column. • False indicates that Table Store does not create an index for the column. • If no indexes exist, you cannot query by the column.
EnableSortAndAgg	Boolean	Specifies whether to enable sorting and aggregation.	<ul style="list-style-type: none"> • True indicates that data can be sorted by using the column. • False indicates that data cannot be sorted by using the column.
Store	Boolean	Specifies whether to store original values in the index.	True indicates that Table Store stores the original values in the column to the index. Therefore, Table Store reads values of the column directly from the index, rather than from the primary table. This optimizes query performance.

Attribute	Type	Option	Description
IsArray	Boolean	Specifies whether the column is an array.	<ul style="list-style-type: none"> • True indicates that the column is an array. Data written to the column must be a JSON array, such as ["a","b","c"]. • You do not need to explicitly specify this parameter for Nested columns because they are arrays. • Array type data can be used in all queries because arrays do not affect queries.

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

Type	Index	EnableSort AndAgg	Store	Array
Long	Supported	Supported	Supported	Supported
Double	Supported	Supported	Supported	Supported
Boolean	Supported	Supported	Supported	Supported
Keyword	Supported	Supported	Supported	Supported
Text	Supported	Not supported	Supported	Supported
Geopoint	Supported	Supported	Supported	Supported
<i>Nested</i>	Required for child fields.	Required for child fields.	Required for child fields.	Nested fields are arrays.

Query parameters and types

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

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

Name	Query	Description
Query by matching all rows	<i>MatchAllQuery</i>	You can use <code>MatchAllQuery</code> to check the total number of rows.
Query by tokenized data	<i>MatchQuery</i>	You can use <code>MatchQuery</code> to tokenize the query data , and query the tokenized data. Logical operator OR applies to tokens.

Name	Query	Description
Query by matched phrases	<i>MatchPhraseQuery</i>	This query is similar to MatchQuery. The matched tokens must be adjacent to each other in the query data.
Query by exact match	<i>TermQuery</i>	You can use TermQuery to match exact strings. Table Store uses exact matches to query data in a table, and does not tokenize the query data.
Query by multiple terms	<i>TermsQuery</i>	This query is similar to TermQuery. You can use TermsQuery to match multiple terms, which is similar to the SQL IN operator.
Query by prefix	<i>PrefixQuery</i>	You can use PrefixQuery to query data in a table by matching a specified prefix.
Query by range	<i>RangeQuery</i>	You can use RangeQuery to query data within a specified range in a table.
Query by wildcards	<i>WildcardQuery</i>	You can use WildcardQuery to query data based on strings that contain one or more wildcards. This query is similar to the SQL LIKE operator.
Query by a combination of filtering conditions	<i>BoolQuery</i>	You can use BoolQuery to combine multiple filtering conditions by using Logical operators, such as AND, OR, and NOT .

Name	Query	Description
Query by matching data within a rectangular geographical area	<i>GeoBoundingBoxQuery</i>	You can use <i>GeoBoundingBoxQuery</i> to specify a rectangular geographical area as a filtering condition in a query. Table Store returns the rows where the value of a field falls within the rectangular geographical area.
Query by matching data within a circular geographical area	<i>GeoDistanceQuery</i>	You can use <i>GeoDistanceQuery</i> to specify a circular geographical area as a filtering condition in a query, which consists of a central point and radius. Table Store returns the rows where the value of a field falls within the circular geographical area.
Query by matching data within a polygonal geographical area	<i>GeoPolygonQuery</i>	You can use <i>GeoPolygonQuery</i> to specify a polygonal geographical area as a filtering condition in a query. Table Store returns the rows where the value of a field falls within the polygonal geographical area.

Pricing

For more information, see [#unique_77](#).

7.3.2 CreateSearchIndex

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

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

Description

Parameters:

- **TableName:** specifies the name of the table for which you want to create a search index.
- **IndexName:** specifies the name of the search index.
- **IndexSchema:** defines the schema of the search index.
 - **IndexSetting**
 - **RoutingFields:** specifies the routing fields. You can specify some primary key columns as routing fields. Table Store distributes data that is written to a search index to different partitions based on the specified routing fields.

The data with the same routing field values is distributed to the same data partition.

- **FieldSchemas**

- **FieldName:** required. This parameter specifies the name of the field that is a column name in the table. The name is of the string type.
- **FieldType:** required. This parameter specifies the type of the field. For more information, see the "Fields" section in [Overview](#).
- **Index:** optional. This parameter specifies whether to create an index for the field. The index is of the Boolean type. Default value: true.
- **IndexOptions:** optional. This parameter specifies whether to store terms such as position and offset in an inverted list. Use the default value in general conditions.
- **EnableSortAndAgg:** optional. This parameter specifies whether to enable sorting and aggregation. This parameter is of the Boolean type. Default value: true.
- **Store:** optional. This parameter specifies whether to store original values in the index to accelerate queries. This parameter is of the Boolean type. Default value: true.

FAQ

How many indexes can be created in a table?

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

• **Method 1: Create a search index for an index field**

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

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

The implementation of this method is similar to that of secondary indexes.

In this case, one index field for one search index brings no benefits to search

indexing but increases costs. Therefore, we recommend that you do not use this method to create a search index.

- **Method 2: Create a search index for multiple index fields**

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

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

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

Limits

1. Timeliness of index creation

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

2. Quantity

For more information, see [Limits](#).

Examples

```
/**
 *Create a search index that contains the Col_Keyword and Col_Long
 columns. Set the type of data in Col_Keyword to KEYWORD. Set the type
 of data in Col_Long to LONG.
 */
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(TABLE_NAME); // Set the table name.
    request.setIndexName(INDEX_NAME); // Set the index name.
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
        new FieldSchema("Col_Keyword", FieldType.KEYWORD) // Set
the field name and field type.
        .setIndex(true) // Set the parameter to true to
enable indexing.
        .setEnableSortAndAgg(true), // Set the parameter
to true to enable sorting and aggregation.
        new FieldSchema("Col_Long", FieldType.LONG)
        .setIndex(true)
        .setEnableSortAndAgg(true)));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); // Use the client to create a
search index.
```

```
}
```

7.3.3 DescribeSearchIndex

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

Description

Name: DescribeSearchIndex

Parameters:

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

Example

```
private static DescribeSearchIndexResponse describeSearchIndex(
    SyncClient client) {
    DescribeSearchIndexRequest request = new DescribeSearchIndexR
    equest();
    request.setTableName(TABLE_NAME); // Set the name of the table.
    request.setIndexName(INDEX_NAME); // Set the name of the index.
    DescribeSearchIndexResponse response = client.describeSearchIndex(
    request);
    System.out.println(response.toJsonize()); // Display the details of
    the response.
    return response;
}
```

7.3.4 ListSearchIndex

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

Description

Name: ListSearchIndex

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

Example

```
private static List<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 Index structures of the specified table.
}
```

7.3.5 DeleteSearchIndex

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

Description

Name: DeleteSearchIndex

Parameters:

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

Example

```
private static void 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 index.
    client.deleteSearchIndex(request); // Use client to delete the
target Search Index structure.
}
```

7.3.6 Array and Nested field types

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

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

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

Array type

Basic Array types, such as:

- **Long Array:** an array of long integers. Format: "[1000, 4, 5555]."
- **Boolean Array:** an array of Boolean values. Format: "[true, false]."
- **Double Array:** an array of floating-point numbers. Format: "[3.1415926, 0.99]."
- **Keyword Array:** an array of strings.

- **Text Array:** an array of text. This type is not common.
- **GeoPoint Array:** an array of geographical locations. Format: "[34.2, 43.0], [21.4, 45.2]."

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

Nested type

A Nested column contains nested documents. One document or one row can contain multiple child documents, and these child documents are saved to the same Nested column. You need to specify the schema of child documents in the Nested column. The structure includes the fields of the child documents and the property of each field. The following example defines the format of a Nested column in Java:

```
// Specify the FieldSchema class for the child documents.
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));

// Set FieldSchema of the child documents as subfieldSchemas of the
// Nested column.
FieldSchema nestedFieldSchema = new FieldSchema("tags", FieldType.
    NESTED)
    .setSubFieldSchemas(subFieldSchemas);
```

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

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

```
[{"tagName":"tag1", "score":0.8}, {"tagName":"tag2", "score":0.2}]
```

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

The Nested type has the following limits:

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

2. The nested query provides lower performance than other types of queries.

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

7.3.7 Sort

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

The Search Index feature supports multiple sorting methods.

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

Table Store supports the following sorting methods:

- **ScoreSort**

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

- **PrimaryKeySort**

Sort the result by the value of a primary key.

- **FieldSort**

Sort the result by the value of a specified field.

- **GeoDistanceSort**

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

7.3.8 Tokenization

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

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

The following tokenization methods are supported:

Methods

Single-word tokenization

- **Name: single_word**

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

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

Alphanumeric characters such as product models cannot be tokenized by this method because there are no spaces or punctuation between letters and numbers. For example, "iPhone6" remains "iPhone6" after tokenization. When querying by `MatchQuery` or `MatchPhraseQuery`, you can retrieve data only by specifying "iphone6" to query. You can set the `delimitWord` parameter to true to separate English letters from numbers. This way, "iphone6" is tokenized into "iphone" and "6".

Delimiter tokenization

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

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

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

**Note:**

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

Minimum semantic unit-based tokenization

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

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

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

Maximum semantic unit-based tokenization

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

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

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

Fuzzy tokenization

- **Name:** fuzzy
- **Applies to:** all natural languages, such as Chinese, English, and Japanese
- **Parameter:**
 - **minChars:** specifies the minimum number of characters for a token. We recommend that you set this value to 2.
 - **maxChars:** specifies the maximum number of characters for a token. We recommend that you set this value to a number smaller than or equal to 7.
- **Limits:**
 - A text field cannot exceed 32 characters in length. Only the first 32 characters of a text field is retained and the characters after the 32nd character are truncated and discarded.

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

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

Comparison

The following table compares the five tokenization methods.

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

7.3.9 MatchAllQuery

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

Example

```
/**
 * Use MatchAllQuery 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 a table. This value is an
     * approximate value when you query a table that contains a large number
     * of rows.
     * To return only the total number of rows without any specific
     * data, you can set Limit to 0. Then, Table Store returns no data in the
     * rows.
     */
    searchQuery.setLimit(0);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
        INDEX_NAME, searchQuery);

    /**
     * Set the total number of matched rows.
     */
    searchQuery.setGetTotalCount(true);
    SearchResponse resp = client.search(searchRequest);
    /**
     * Check whether Table Store returns matched data from all
     * partitions. When the value of isSuccess is false, Table Store may
     * fail to query some partitions and return a part of data.
     */
    if (! resp.isSuccess()) {
        System.out.println("NotAllSuccess!") ;
    }
    System.out.println("IsAllSuccess: " + resp.isSuccess());
    System.out.println("TotalCount: " + resp.getTotalCount()); // The
    total number of rows.
    System.out.println(resp.getRequestId());
}
```

7.3.10 MatchQuery

You can use MatchQuery to query data in the fields of Text type in full-text search scenarios. Table Store tokenizes the value of Text type in the index and the target

value that you specify for the MatchQuery type based on your configuration.

Therefore, Table Store can match tokenized terms in a query.

For example, the title field value in a row is "Hangzhou West Lake Scenic Area".

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

Parameters

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

Example

```
/**
 * Search the table for rows where the value of Col_Keyword matches "
 hangzhou". Table Store returns matched rows and the total number of
 matched rows.
 * @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"); // Set the name of the
    field that you want to match.
    matchQuery.setText("hangzhou"); // Set the value that you want to
    match.
    searchQuery.setQuery(matchQuery);
    searchQuery.setOffset(0); // Set Offset to 0.
    searchQuery.setLimit(20); // Set Limit to 20 to return 20 rows or
    fewer.
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount());
    System.out.println("Row: " + resp.getRows()); // If you do not set
    columnsToGet, Table Store only returns primary keys by default.

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
    ColumnsToGet();
}
```



```

        columnsToGet.setReturnAll(true); // Set columnsToGet to true to
        return all columns.
        searchRequest.setColumnsToGet(columnsToGet);

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

```

7.3.11 MatchPhraseQuery

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

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

Parameters

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

Example

```

/**
 * Search the table for rows where the value of Col_Text matches "
 * hangzhou shanghai." Table Store returns the total number of rows that
 * match the phrase as a whole and matched rows in this query.
 * @param client
 */
private static void matchPhraseQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    MatchPhraseQuery matchPhraseQuery = new MatchPhraseQuery(); // Set
    the query type to MatchPhraseQuery.
    matchPhraseQuery.setFieldName("Col_Text"); // Set the field that
    you want to match.
    matchPhraseQuery.setText("hangzhou shanghai"); // Set the value
    that you want to match.
    searchQuery.setQuery(matchPhraseQuery);
    searchQuery.setOffset(0); // Set Offset to 0.
    searchQuery.setLimit(20); // Set Limit to 20 to return 20 rows or
    fewer.
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
}

```

```
        System.out.println("TotalCount: " + resp.getTotalCount());
        System.out.println("Row: " + resp.getRows()); // Return primary
        keys only by default.

        SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
        ColumnsToGet();
        columnsToGet.setReturnAll(true); // Set columnsToGet to true to
        return all columns.
        searchRequest.setColumnsToGet(columnsToGet);

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

7.3.12 TermQuery

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

Parameters

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

Example

```
/**
 * Search the table for rows where the value of Col_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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
    ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set columnsToGet to true to
    return all columns.
    searchRequest.setColumnsToGet(columnsToGet);

    SearchResponse resp = client.search(searchRequest);
}
```

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

7.3.13 TermsQuery

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

Parameters

fieldName: the name of the target field.

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

Example

```
/**
 * Search the table for rows where the value of Col_Keyword exactly
 * matches "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"); // Set the name of the
    field that you want to match.
    termsQuery.addTerm(ColumnValue.fromString("hangzhou")); // Set the
    value that you want to match.
    termsQuery.addTerm(ColumnValue.fromString("xi'an")); // Set the
    value that you want to match.
    searchQuery.setQuery(termsQuery);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
    ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set columnsToGet to true to
    return all columns.
    searchRequest.setColumnsToGet(columnsToGet);

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

```
}
```

7.3.14 PrefixQuery

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

Parameters

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

Example

```
/**
 * Search the table for rows where the value of Col_Keyword contains
 * the prefix that exactly matches "hangzhou".
 * @param client
 */
private static void prefixQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    PrefixQuery prefixQuery = new PrefixQuery(); // Set the query type
    to PrefixQuery.
    prefixQuery.setFieldName("Col_Keyword");
    prefixQuery.setPrefix("hangzhou");
    searchQuery.setQuery(prefixQuery);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
    ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set columnsToGet to true to
    return all columns.
    searchRequest.setColumnsToGet(columnsToGet);

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

7.3.15 RangeQuery

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

Parameters

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

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

Example

```
/**
 * Search the table for rows where the value of Col_Long is greater
 * than 3. Table Store sorts these rows by Col_Long in descending order.
 * @param client
 */
private static void rangeQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    RangeQuery rangeQuery = new RangeQuery(); // Set the query type to
    RangeQuery.
    rangeQuery.setFieldName("Col_Long"); // Set the name of the
    target field.
    rangeQuery.greaterThan(ColumnValue.fromLong(3)); // Specify the
    range of the value of the field. The required value is larger than 3.
    searchQuery.setQuery(rangeQuery);

    // Sort the result 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
    )));

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);

    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount()); // The
    total number of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());

    /**
     * You can specify a value for SearchAfter to start a new query
     * . For example, you can set SearchAfter to 5 and sort the result by
     * Col_Long in descending order. Then, you retrieve the rows that follow
     * the row whose Col_Long is equal to 5. This is similar to the method
     * where you specify that the value of Col_Long is smaller than 5.
     */
    searchQuery.setSearchAfter(new SearchAfter(Arrays.asList(
    ColumnValue.fromLong(5))));
    searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME,
    searchQuery);
    resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount()); // The
    total number of matched rows instead of the number of returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

7.3.16 WildcardQuery

You can use WildcardQuery to query data that matches wildcard characters. You can specify a value you want to match as a string that consists of one or more

wildcard characters. An asterisk (*) is interpreted as a number of characters or an empty string. A question mark (?) is interpreted as any single character. For example, when you search the string "table*e", you can retrieve query results such as "tablestore".

Parameters

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

Example

```
/**
 * Search the table for rows where the value of Col_Keyword 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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set columnsToGet to true to
return all columns.
    searchRequest.setColumnsToGet(columnsToGet);

    SearchResponse resp = client.search(searchRequest);

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

7.3.17 BoolQuery

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

You can combine these subqueries in different ways. If you specify these subqueries as mustQueries, Table Store returns the result that matches all these subqueries. If

you specify these subqueries as `mustNotQueries`, Table Store returns the result that matches none of these subqueries.

Parameter

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

Examples

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

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

    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * Create a query of BoolQuery type where the result meets
         * Conditions 1 and 2 at the same time.
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustQueries(Arrays.asList(rangeQuery, matchQuery
    ));
        searchQuery.setQuery(boolQuery);
        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
        INDEX_NAME, searchQuery);
    }
```

```

        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());
    }

    {
        /**
         * Create a query of BoolQuery type where the result meets at
least one of Condition 1 and 2.
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setShouldQueries(Arrays.asList(rangeQuery,
matchQuery));
        boolQuery.setMinimumShouldMatch(1); // Specify that the result
meets at least one of the conditions.
        searchQuery.setQuery(boolQuery);
        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
        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());
    }
}

```

7.3.18 Nested query

This topic describes how to query nested fields. You can use a nested query to query the data of nested fields.

You must use the NestedQuery operation to wrap the nested field before its query data can be used. In NestedQuery, you must specify a subquery of any type and the path of the nested field.

You can only query fields of the nested type.

Common fields and nested fields can be queried simultaneously within a request.

Parameters

- **path:** the tree path of the nested fields content. For example, news.title indicates the title nested within the news field.
- **query:** the query to perform on the child field of the nested field. It can be of any type.

Examples

```

/**
 * The NESTED column contains nested_1 and nested_2. You need
to search the col_nested.nested_1 column for data that matches "
tablestore".
 * @param client

```



```

*/
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
    field.
    TermQuery termQuery = new TermQuery(); // Specify a subquery for
    NestedQuery.
    termQuery.setFieldName("col_nested.nested_1"); // Set the name
    of the field that you want to match. The field name must contain the
    prefix of the Nested column.
    termQuery.setTerm(ColumnValue.fromString("tablestore")); // Set
    the value that you want to match.
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    searchQuery.setGetTotalCount(true);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
    ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set ReturnAll to true to
    return all 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());
}

```

7.3.19 GeoDistanceQuery

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

Parameters

- **fieldName:** the name of the target field.
- **centerPoint:** the central coordinate point that consists of latitude and longitude values.
- **distanceInMeter:** the distance from the central point. This is a value of **Double** type. Unit: meters.

Example

```

/**
 * Search the table for rows where the value of Col_GeoPoint falls
 * within a specified distance from a specified central point.
 * @param client

```

```

    */
    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 coordinates for
        a central point.
        geoDistanceQuery.setDistanceInMeter(10000); // You can specify 10,
        000 meters or less as the distance from the central point.
        searchQuery.setQuery(geoDistanceQuery);

        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
        INDEX_NAME, searchQuery);

        SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
        ColumnsToGet();
        columnsToGet.setColumns(Arrays.asList("Col_GeoPoint")); //Specify
        Col_GeoPoint as the column that you want to return.
        searchRequest.setColumnsToGet(columnsToGet);

        SearchResponse resp = client.search(searchRequest);
        System.out.println("TotalCount: " + resp.getTotalCount()); // The
        total number of matched rows instead of the number of returned rows.
        System.out.println("Row: " + resp.getRows());
    }

```

7.3.20 GeoBoundingBoxQuery

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

Parameters

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

Example

```

/**
 * The data type of Col_GeoPoint is Geopoint. You can obtain the rows
 * where the value of Col_GeoPoint falls within a geographic rectangular
 * area. For the geographic rectangular area, the upper-left vertex is "
 * 10,0" and the lower-right vertex is "0,10".
 * @param client
 */
    public static void geoBoundingBoxQuery(SyncClient client) {
        SearchQuery searchQuery = new SearchQuery();
        GeoBoundingBoxQuery geoBoundingBoxQuery = new GeoBoundingBoxQuery
        (); // Set the query type 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 vertex of the geographic rectangular area.
        geoBoundingBoxQuery.setBottomRight("0,10"); // Specify coordinates
        for the lower-right vertex of the geographic rectangular area.
        searchQuery.setQuery(geoBoundingBoxQuery);

        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
        INDEX_NAME, searchQuery);

        SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
        ColumnsToGet();
        columnsToGet.setColumns(Arrays.asList("Col_GeoPoint")); //Specify
        Col_GeoPoint as the column that you want to return.
        searchRequest.setColumnsToGet(columnsToGet);

        SearchResponse resp = client.search(searchRequest);
        System.out.println("TotalCount: " + resp.getTotalCount()); // The
        total number of matched rows instead of the number of returned rows.
        System.out.println("Row: " + resp.getRows());
    }

```

7.3.21 GeoPolygonQuery

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

Parameters

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

Example

```

/**
 * Search the table for rows where the value of Col_GeoPoint falls
 * within a specified geographic polygon area.
 * @param client
 */
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 coordinates for vertices of the geographic polygon.
    searchQuery.setQuery(geoPolygonQuery);

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
    ColumnsToGet();
    columnsToGet.setColumns(Arrays.asList("Col_GeoPoint")); //Specify
    Col_GeoPoint as the column that you want to return.
}

```

```
searchRequest.setColumnsToGet(columnsToGet);

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

7.3.22 ExistQuery

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



Note:

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

Parameter

fieldName: the column name

Examples

```
/**
 * Use ExistQuery to query the rows in which the value of the address
 * column is not null.
 * @param client
 */
private static void termsQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    ExistsQuery existQuery = new ExistsQuery(); // Set the query type
    to ExistsQuery.
    existQuery.setFieldName("address");
    searchQuery.setQuery(termsQuery);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
    INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
    ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set ReturnAll to true to
    return all 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());
}
```

```
}
```

7.3.23 Statistics

This topic describes statistical operations used in search index-based queries.

Types

- **Minimum value**

- **Definition:** Query the minimum value of a field. This query is equivalent to the SQL MIN function. If a row does not include a field value, the row is not included in the statistics. However, you can set a default value for rows that do not have any value for the specified field.
- The following table describes the parameters.

Parameter	Description
aggregationName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name .
fieldName	The name of the field for which to obtain statistics . Only fields of the long and double types are supported.
missing	The default value of rows that do not have any value for the specified field. If a row is missing a value for a specified field and this parameter is not specified, the row is ignored. If this parameter is set, the value of this parameter is used as the field value of the row.

- **Java example**

```
/**
 * The price of each product is listed in the product table. Query
 * the minimum price among the products produced in Zhejiang.
 * The equivalent SQL statement: SELECT min (column_price) FROM
 * product where place_of_production = "Zhejiang";
 */
public void min(SyncClient client) {
    // Create a query
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "Zhejiang"))
        )
}
```

```

        .limit(0)    // If you only want to obtain the
        aggregation results, you can set the number of matched results to
        be returned to 0 to reduce the response time.
        .addAggregation(AggregationBuilders.min("min_agg_1", "column_price").missing(100))
        .build()
    .build();
    //Execute the query
    SearchResponse resp = client.search(searchRequest);
    //Obtain the aggregation results
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("min_agg_1").getValue());
}

```

- **Maximum value**

- **Definition:** Query the maximum value of a field. This query is equivalent to the SQL MAX function. If a row does not include a field value, the row is not included in the statistics. However, you can set a default value for rows that do not have any value for the specified field.

- **Parameters**

Parameter	Description
aggregationName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name .
fieldName	The name of the field for which to obtain statistics . Only fields of the long and double types are supported.
missing	The default value of rows that do not have any value for the specified field. If a row is missing a value for a specified field and this parameter is not specified, the row is ignored. If this parameter is set, the value of this parameter is used as the field value of the row.

- **Java example**

```

/**
 * The price of each product is listed in the product table. Query
 * the maximum price among the products produced in Zhejiang.
 * The equivalent SQL statement: SELECT max (column_price) FROM
 * product where place_of_production = "Zhejiang";
 */
public void max(SyncClient client) {
    // Create a query
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(

```

```

        SearchQuery.newBuilder()
            .query(QueryBuilders.term("place_of_production", "
Zhejiang"))
            .limit(0)    // If you only want to obtain the
                        // aggregation results, you can set the number of matched results to
                        // be returned to 0 to reduce the response time.
            .addAggregation(AggregationBuilders.max("max_agg_1", "column_price").missing(0))
            .build()
        .build();
        //Execute the query
        SearchResponse resp = client.search(searchRequest);
        //Obtain the aggregation results
        System.out.println(resp.getAggregationResults().getAsMaxAggregationResult("max_agg_1").getValue());
    }

```

- **Sum**

- **Definition:** Query the total value of all rows for a numeric field. This query is equivalent to the SQL SUM function. If a row does not include a field value, the row is not included in the statistics. However, you can set a default value for rows that do not have any value for the specified field.
- **Parameters**

Parameter	Description
aggregationName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name .
fieldName	The name of the field for which to obtain statistics . Only fields of the long and double types are supported.
missing	The default value of rows that do not have any value for the specified field. If a row is missing a value for a specified field and this parameter is not specified, the row is ignored. If this parameter is set, the value of this parameter is used as the field value of the row.

- **Java example**

```

/**
 * The sales volume of each product is listed in the product table
 * . Query the total number of the sold products that are produced in
 * Zhejiang. Set the value of missing to 10.
 * The equivalent SQL statement: SELECT sum (column_price) FROM
 * product where place_of_production = "Zhejiang";
 */
public void sum(SyncClient client) {
    // Create a query

```

```

        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .searchQuery(
                SearchQuery.newBuilder()
                    .query(QueryBuilders.term("place_of_production", "
Zhejiang"))
                    .limit(0) // If you only want to obtain the
aggregation results, you can set the number of matched results to
be returned to 0 to reduce the response time.
                    .addAggregation(AggregationBuilders.sum("sum_agg_1
", "column_number").missing(10))
                    .build())
            .build();
        //Execute the query
        SearchResponse resp = client.search(searchRequest);
        //Obtain the aggregation results
        System.out.println(resp.getAggregationResults().getAsSumAg
gregationResult("sum_agg_1").getValue());
    }

```

- **Average**

- **Definition:** Query the average value of all rows for a numeric field. This query is equivalent to the SQL AVG function. If a row does not include a field value, the row is not included in the statistics. However, you can set a default value for rows that do not have any value for the specified field.

- **Parameters**

Parameter	Description
aggregationName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name .
fieldName	The name of the field for which to obtain statistics . Only fields of the long and double types are supported.
missing	The default value of rows that do not have any value for the specified field. If a row is missing a value for a specified field and this parameter is not specified, the row is ignored. If this parameter is set, the value of this parameter is used as the field value of the row.

- **Java example**

```

/**
 * The sales volume of each product is listed in the product table
 * . Query the average price of the products produced in Zhejiang.
 * The equivalent SQL statement: SELECT avg (column_price) FROM
product where place_of_production = "Zhejiang";

```



```

*/
public void avg(SyncClient client) {
    // Create a query
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "
Zhejiang"))
                .limit(0)    // If you only want to obtain the
aggregation results, you can set the number of matched results to
be returned to 0 to reduce the response time.
                .addAggregation(AggregationBuilders.avg("avg_agg_1
", "column_number")))
            .build())
        .build();
    //Execute the query
    SearchResponse resp = client.search(searchRequest);
    //Obtain the aggregation results
    System.out.println(resp.getAggregationResults().getAsAvgAg
gregationResult("avg_agg_1").getValue());
}

```

- **Count**

- **Definition:** Query the total number of values for a field. This query is equivalent to the SQL COUNT function. If the field value in a row does not exist, the row is not included in the statistics.



Note:

The current count operation does not support the COUNT (*) function. To count the rows in an index or the matched rows in a query, use query operations and set the setGetTotalCount parameter to true in the query.

- **Parameters**

Parameter	Description
aggregationName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name .
fieldName	The name of the field for which to obtain statistics . Only fields of the long, double, Boolean, keyword , and geo_point types are supported.

- **Java example**

```

/**
 * The punishment records of each merchant is recorded in the
merchant table. Query the number of merchants in Zhejiang who have
punishment records. (Assume that merchants with no punishment
records do not have a value for the specified field.)

```

```

* The equivalent SQL statement: SELECT count(column_history) FROM
product where place_of_production="Zhejiang";
*/
public void count(SyncClient client) {
    // Create a query
    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
    SearchResponse resp = client.search(searchRequest);
    //Obtain the aggregation results
    System.out.println(resp.getAggregationResults().getAsCount
AggregationResult("count_agg_1").getValue());
}

```

- Distinct count

■ **Definition:** Query the number of distinct values for a field. This query is equivalent to the SQL COUNT (distinct) function. You can set a default value for rows that do not have any value for the specified field.

■ Parameters

Parameter	Description
aggregationName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name.
fieldName	The name of the field for which to obtain statistics. Only fields of the long, double, Boolean, keyword, and geo_point types are supported.
missing	The default value of rows that do not have any value for the specified field. If a row is missing a value for a specified field and this parameter is not specified, the row is ignored. If this parameter is set, the value of this parameter is used as the field value of the row.

■ Java example

```

/**
 * Query the number of distinct provinces from which all
products come.
 * The equivalent SQL statement: SELECT count(distinct
column_place) FROM product;

```

```
*/
public void distinctCount(SyncClient client) {
    // Create a query
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.distinctCount("dis_count_agg_1", "column_place"))
                .build()
        )
        .build();
    //Execute the query
    SearchResponse resp = client.search(searchRequest);
    //Obtain the aggregation results
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("dis_count_agg_1").getValue());
}
```

Multiple statistics

Each request supports multiple statistics.

Java example

```
public void multipleAggregation(SyncClient client) {
    // Create a query
    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
    SearchResponse resp = client.search(searchRequest);
    //Obtain the results of the first aggregation
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").getValue());
    //Obtain the results of the second aggregation
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").getValue());
    //Obtain the results of the third aggregation
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("name3").getValue());
}
```

}

7.3.24 Aggregation

This topic describes aggregation operations in search index-based queries.

Types

- **Group by field values**
 - **Definition:** Group query results based on values for a field. Same values are grouped together. The value of each group and the number of corresponding values are returned. Errors may occur when the number of values in a group is large.
 - **Parameters**

Parameter	Description
groupByName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name.
fieldName	The name of the field to be aggregated. Only fields of the keyword, long, double, and bool types are supported. [DO NOT TRANSLATE]
groupBySorter	<p>Add sorting rules for items in a group. By default, group items are sorted in descending order. When you set multiple sorting rules, items are sorted in the order in which the rules were added. Supported parameters are as follows:</p> <ul style="list-style-type: none"> ■ Sort by parameter keys in alphabetical order. ■ Sort by parameter keys in reverse alphabetical order. ■ Sort by the number of rows in ascending order. ■ Sort by the number of rows in descending order. ■ Sort by the values of sub-aggregations in ascending order. ■ Sort by the values of sub-aggregations in descending order.
size	The number of returned groups.

Parameter	Description
subAggregation and subGroupBy	<p>Add sub-aggregations. Sub-aggregations perform additional aggregation operations on the group content.</p> <p>Example:</p> <ul style="list-style-type: none"> ■ Scenario <p>Query the number of products in each category and the maximum and minimum product prices in each category.</p> ■ Method <p>Group query results by product categories, and then add two sub-aggregations to obtain the maximum and minimum product prices in each category.</p> ■ Result <ul style="list-style-type: none"> ■ Fruits: 5. The minimum price is CNY 3, and the maximum price is CNY 15. ■ Toiletries: 10. The minimum price is CNY 1, and the maximum price is CNY 98. ■ Electronic devices: 3. The minimum price is CNY 2,300, and the maximum price is CNY 8,699. ■ Other products: 15. The minimum price is CNY 80, and the maximum price is CNY 1,000.

- **Java example**

```

/**
 * Query the number of products in each category and the maximum
 * and minimum product prices in each category.
 * Example of returned results: "Fruits: 5. The minimum price
 * is CNY 3, and the maximum price is CNY 15. Toiletries: 10.
 * The minimum price is CNY 1, and the maximum price is CNY 98.
 * Electronic devices: 3. The minimum price is CNY 2,300, and the
 * maximum price is CNY 8,699. Other products: 15. The minimum price
 * is CNY 80,
 * and the maximum price is CNY 1,000.
 */
public void groupByField(SyncClient client) {
    // Create a query
    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
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results
    for (GroupByFieldResultItem item : resp.getGroupByResults().
getAsGroupByFieldResult("name1").getGroupByFieldResultItems()) {
        // Value
        System.out.println(item.getKey());
        // Count
        System.out.println(item.getRowCount());
        // Minimum price
        System.out.println(item.getSubAggregationResults().
getAsMinAggregationResult("subName1").getValue());
        // Maximum price
        System.out.println(item.getSubAggregationResults().
getAsMaxAggregationResult("subName2").getValue());
    }
}

```

- **Group by range**

- **Definition:** Group query results based on value ranges of a field. Field values that fall within a certain range are grouped together. The number of items in each range is returned. For example, group the sales volume of a product by [0 , 1000), [1000, 10000), and [10000, ∞) ranges to obtain the sales volume of each range.
- **Parameters**

Parameter	Description
groupByName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name.
fieldName	The name of the field to be aggregated. Only fields of the long and double types are supported.
range[double_fro m, double_to)	Add ranges for grouping. The range can start from the Double.MIN_VALUE and end at the Double.MAX_VALUE.

Parameter	Description
subAggregation and subGroupBy	Add sub-aggregations. Sub-aggregations perform additional aggregation operations on the group content . For example, after you group query results by sales volume and then by province, you can obtain which province has the largest proportion in a certain range of sales volume. You must add a GroupByField sub-aggregation in the GroupByRange aggregation to execute this query.

- **Java example**

```
/**
 * Group sales volume by [0, 1000), [1000, 5000), and [5000, ∞)
 * ranges to obtain the sales volume in each range.
 */
public void groupByRange(SyncClient client) {
    // Create a query
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders.groupByField()
                    .groupByRange("name1", "column_number")
                    .addRange[0, 1000)
                    .addRange[1000, 5000)
                    .addRange[5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    // Execute the query
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results
    for (GroupByRangeResultItem item : resp.getGroupByResults().
        getAsGroupByRangeResult("name1").getGroupByRangeResultItems()) {
        // Range
        System.out.println(item.getKey());
        // Count
        System.out.println(item.getRowCount());
    }
}
```

• **Group by geographical distance**

- **Definition:** Group query results based on their geographical distances to a central point. Distance differences that fall within a certain range are grouped together. The number of items in each range is returned. For example, group people based on their geographical distances to a Wanda Plaza to obtain the

number of people in each distance range. Group the distance differences by [0, 1000m) [1000m, 5000m), and [5000m, ∞) ranges.

- Parameters

Parameter	Description
groupByName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name.
fieldName	The name of the field to be aggregated. Only fields of the <code>geo_point</code> type are supported.
origin(double lat, double lon)	lat indicates the latitude of the central point, and lon indicates the longitude of the central point.
range(double_from, double_to)	Add ranges for grouping. Unit: m. The range can start from the <code>Double.MIN_VALUE</code> and end at the <code>Double.MAX_VALUE</code> .
subAggregation and subGroupBy	Add sub-aggregations. Sub-aggregations perform additional aggregation operations on the group content.

- Java example

```
/**
 * Group people based on their geographical distances to a Wanda
 * Plaza to obtain the number of people in each distance range. Group
 * the distance differences by [0,1000m) [1000m, 5000m), and [5000m,
 *  $\infty$ ) ranges.
 */
public void groupByGeoDistance(SyncClient client) {
    // Create a query
    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
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results
    for (GroupByGeoDistanceResultItem item : resp.getGroupByResults().getAsGroupByGeoDistanceResult("name1").getGroupByGeoDistanceResultItems()) {
        // Range
    }
}
```



```

        System.out.println(item.getKey());
        // Count
        System.out.println(item.getRowCount());
    }
}

```

- **Group by filter**

- **Definition:** Filter the query results and group them together to obtain the number of items matching each filter. Results are returned in the order in which the filters are added. For example, you can add the following three filters to obtain the number of items matching each filter: sales volume exceeds 100, production place is Zhejiang, and description contains Hangzhou

- **Parameters**

Parameter	Description
groupByName	The name given to the aggregation to distinguish it from other aggregation operations. You can find the required aggregation results based on its name.
filter	The added filters of a query. Filters are added using QueryBuilders.
subAggregation and subGroupBy	Add sub-aggregations. Sub-aggregations perform additional aggregation operations on the group content.

- **Java example**

```

/**
 * Filter the query results. For example, add the following three
 * filters to obtain the number of items matching each filter: sales
 * volume exceeds 100, production place is Zhejiang, and description
 * contains Hangzhou.
 */
public void groupByFilter(SyncClient client) {
    // Create a query
    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
        SearchResponse resp = client.search(searchRequest);
        // Obtain the aggregation results in the order of addFilter.
        for (GroupByFilterResultItem item : resp.getGroupByResults().
            getAsGroupByFilterResult("name1").getGroupByFilterResultItems()) {
            // Count
            System.out.println(item.getRowCount());
        }
    }
}

```

Multiple aggregations

A request supports multiple aggregations.



Note:

Implementing multiple complicated aggregations at the same time may cause a long response time.

Java example:

```

public void multipleGroupBy(SyncClient client) {
    // Create a query
    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
    SearchResponse resp = client.search(searchRequest);
    // Obtain the results of the first aggregation
    System.out.println(resp.getAggregationResults().getAsMinAg
gregationResult("name1").getValue());
    // Obtain the results of the second aggregation
    System.out.println(resp.getAggregationResults().getAsSumAg
gregationResult("name2").getValue());
    // Obtain the results of the third aggregation
    System.out.println(resp.getAggregationResults().getAsDisti
nctCountAggregationResult("name3").getValue());
    // Obtain the results of the fourth aggregation
    for (GroupByFieldResultItem item : resp.getGroupByResults().
        getAsGroupByFieldResult("name4").getGroupByFieldResultItems()) {
        // Key
        System.out.println(item.getKey());
        // Count
    }
}

```

```

        System.out.println(item.getRowCount());
    }
    // Obtain the results of the fifth aggregation
    for (GroupByRangeResultItem item : resp.getGroupByResults().
getAsGroupByRangeResult("name4").getGroupByRangeResultItems()) {
        // Count
        System.out.println(item.getRowCount());
    }
}

```

Nesting

GroupBy type aggregations support nesting. You can add sub-aggregations and GroupBy type sub-aggregations to a GroupBy type aggregation. GroupBy type aggregations can contain endless levels for nesting. However, to ensure the performance and reduce the complexity of aggregations, you are allowed only to set a certain number of levels for nesting.

Nested aggregations are used to perform additional aggregation operations within a group. A two-level nested aggregation is used as an example:

- **GroupBy + SubGroupBy:** Group items by province and then by city to obtain data for each city in each province.
- **GroupBy + SubAggregation:** Group items by province to obtain the maximum value of an indicator for each province.

Java example:

```

/**
 * Nested aggregations example: Two aggregations and one GroupByField
 * are added to the outermost level, and two aggregations and one
 * GroupByRange are added to the GroupByField.
 */
public void subGroupBy(SyncClient client) {
    // Create a query
    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
    SearchResponse resp = client.search(searchRequest);
    // The aggregation results of the first level
    AggregationResults aggResults = resp.getAggregationResults();
    System.out.println(aggResults.getAsMinAggregationResult("name1").
getValue());
    System.out.println(aggResults.getAsMaxAggregationResult("name2").
getValue());

    // Retrieve the GroupByField results of the first level and the
    results of the aggregations nested in the GroupByField.
    GroupByFieldResult results = resp.getGroupByResults().getAsGroup
ByFieldResult("someName1");
    for (GroupByFieldResultItem item : results.getGroupByFieldResul
tItems()) {
        System.out.println("Count" + item.getRowCount());
        System.out.println("key:" + item.getKey());

        //Retrieve the results of sub-aggregations
        //The results of the SubAggregation: min
        System.out.println(item.getSubAggregationResults().getAsMaxAg
gregationResult("subName1"));
        //The results of the SubAggregation: max
        System.out.println(item.getSubAggregationResults().getAsSumAg
gregationResult("subName2"));
        //The results of the SubGroupBy: GroupByRange
        GroupByRangeResult subResults = resp.getGroupByResults().
getAsGroupByRangeResult("subName3");
        for (GroupByRangeResultItem subItem : subResults.getGroupBy
RangeResultItems()) {
            System.out.println("Count:" + subItem.getRowCount());
            System.out.println("key:" + subItem.getKey());
        }
    }
}

```

7.4 Limits

This topic describes the limits when using search index.

Mapping

Item	Limit	Description
Index fields	200	The maximum number of fields that can be indexed.
EnableSortAndAgg fields	100	The maximum number of fields that can be sorted and aggregated.
Nested levels	1	The maximum number of nested levels.

Item	Limit	Description
Nested fields	25	The maximum number of nested fields.
Total length of primary key columns	1,000	The sum of all primary key columns can be up to 1,000 Bytes in length.
Total length of strings in primary key columns	1,000	The sum of strings in all primary key columns can be up to 1,000 Bytes in length.
String length in each attribute column (keyword index)	4 KB	None
String length in each attribute column (text index)	2 MB	Same as the length limit of attribute columns in tables.
String length of a query that contains wildcards	10	The string length of a query that contains wildcards can be up to 10 characters.

Search

Type	Item	Limit	Description
General limits	offset + limit	2,000	To read more than 2,000 rows, you must specify the next_token parameter.
	timeout	10s	None
	Capacity unit (CU)	100,000	<ul style="list-style-type: none"> This limit does not apply to scanning and analysis requests. If your business requirements exceed this limit, submit a ticket.

Type	Item	Limit	Description
	QPS	100,000	<ul style="list-style-type: none"> • The upper limit for lightweight transaction processing is 100,000 queries per second (QPS). • Each index is allowed to take up to 8 vCPU for analytical queries or text queries because each request takes a long time. • The preceding limits are fixed by default. If your business requirements exceed the default limits, submit a ticket.
Aggregation	Aggregations at the same level	5	The number of aggregations is recalculated each time you add a new aggregation to SubGroupBy.
	GroupBy type aggregations at the same level	5	The number of GroupBy type aggregations is recalculated each time you add a new GroupBy type aggregation to SubGroupBy.

Type	Item	Limit	Description
	Nested levels for a GroupBy type aggregation	3	The GroupBy type aggregation is included in the calculation of the number of nested levels.
	Filters in a GroupByFilter aggregation	10	None
	Groups returned by a GroupByField aggregation	200	None
	Ranges in a GroupByRange aggregation	100	None
	Ranges in a GroupByGeo distance aggregation	10	None

Index

Item	Limit	Description
Rate	50,000 rows /s	<ul style="list-style-type: none"> Table Store requires several minutes for load balancing when writing data to a table for the first time or when a large amount of data is required to be written in a short span of time. Text field-based indexing is limited to 10,000 rows per second because this indexing consumes more CPU resources for tokenization. If your business requirements exceed this limit, submit a ticket.
Synchronization latency	10s	<ul style="list-style-type: none"> The value is less than 10s when the writing rate is steady. The synchronization latency is within one minute in most cases. New indexes need to be initialized. This process takes up to one minute.

Item	Limit	Description
Number of rows	10 billion	If your business requirements exceed this limit, submit a ticket.
Total size	10 TB	If your business requirements exceed this limit, submit a ticket.

Other limits

Item	Value
Supported regions	China (Beijing), China (Shanghai), China (Hangzhou), China (Shenzhen), Singapore, India (Mumbai), China (Hong Kong), and China (Zhangjiakou-Beijing Winter Olympics)
Regions awaiting release	US (Silicon Valley)

**Note:**

If your business requirements exceed the default limits, submit a ticket. Describe the scenario, limit items, requirements, and reasons in the ticket. Your requirements will be considered during future development.

8 Global secondary index

8.1 Overview

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

Terms

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

Limits

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

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

Notes

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

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

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

- When you create an index for the `Defined0` column, Table Store generates the index that has Primary keys `Defined0`, `PK0`, and `PK1`.
- When you create an index for the `PK1` column, Table Store generates the index that has Primary keys `PK1` and `PK0`.
- You can specify pre-defined columns as attribute columns in the primary table. When you specify a pre-defined attribute as an indexed attribute column, you can search this index for the attribute value instead of searching the primary table. However, this increases storage costs. If you do not specify a pre-defined attribute as an indexed attribute column, you have to search the primary table. You can choose between these methods based on your requirements.
- We recommend that you do not specify a column related to the time or date as the first primary key column of an index. This type of column may slow down index updates. We recommend that you hash the column related to the time or date and create an index for the hashed column. To solve related issues, use DingTalk to request technical support.
- We recommend that you do not define an attribute of low cardinality, even an attribute that contains enumerated values, as the first primary key column of an index. For example, the `gender` attribute restricts the horizontal scalability of the index and leads to poor write performance.

8.2 Introduction

A global secondary index in Tablestore has the following features:

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

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

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

8.3 Scenarios

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

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

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

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

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

If you have the following query requirements:

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

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

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

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

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

    rangeRowQueryCriteria.setMaxVersions(1);

    String strNum = String.format("%d", cellNumber);
```

```

        System.out.println("A cell number " + strNum + "makes the
        following calls:");
        while (true) {
            GetRangeResponse getRangeResponse = client.getRange(new
            GetRangeRequest(rangeRowQueryCriteria));
            for (Row row : getRangeResponse.getRows()) {
                System.out.println(row);
            }

            // If the value of nextStartPrimaryKey is not null, you can
            continue to read data from the base table.
            if (getRangeResponse.getNextStartPrimaryKey() != null) {
                rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
                getRangeResponse.getNextStartPrimaryKey());
            } else {
                break;
            }
        }
    }
}

```

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

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

IndexOnBeCalledNumber:

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



Note:

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

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

```

private static void getRangeFromIndexTable(SyncClient client, long
cellNumber) {

```

```

RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX0_NAME);

// You can specify primary keys.
PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.fromLong(cellNumber));
startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MIN);
startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MIN);
rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());

// You can specify primary keys.
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("A cell number" + strNum + "was called by the following numbers");
while (true) {
    GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQueryCriteria));
    for (Row row : getRangeResponse.getRows()) {
        System.out.println(row);
    }

    // If the value of nextStartPrimaryKey is not null, you can continue to read data from the base table.
    if (getRangeResponse.getNextStartPrimaryKey() != null) {
        rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextStartPrimaryKey());
    } else {
        break;
    }
}

```



```
}
```

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

This query specifies both `BaseStationNumber` and `StartTime` as conditions.

Therefore, you can create a compound index on the `BaseStationNumber` and `StartTime`.

`IndexOnBaseStation1:`

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

You can query the `IndexOnBaseStation1` index.

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX1_NAME);

    // You can specify primary keys.
    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());

    // You can specify primary keys.
    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(endPrimary
KeyBuilder.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 as follows:");
        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, you can
continue to read data from the base table.
            if (getRangeResponse.getNextStartPrimaryKey() != null) {
                rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
            } else {
                break;
            }
        }
    }
}

```

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

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

```

private static void getRowFromIndexAndMainTable(SyncClient client,
                                                long baseStatio
nNumber,
                                                long startTime,
                                                long endTime,
                                                String colName) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX1_NAME);

    // You can specify primary keys.
    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(startPrima
ryKeyBuilder.build());

    // You can specify primary keys.

```

```

        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(endPrimary
KeyBuilder.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 list of calls forwarded by the base
station" + strBaseStationNum + "from" + strStartTime + "to" +
strEndTime + "is listed as follows:");
        while (true) {
            GetRangeResponse getRangeResponse = client.getRange(new
GetRangeRequest(rangeRowQueryCriteria));
            For (Row row: fig. getrows ()) {
                PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
                PrimaryKeyColumn mainCalledNumber = curIndexPrimaryKey.
getPrimaryKeyColumn(PRIMARY_KEY_NAME_1);
                PrimaryKeyColumn callStartTime = curIndexPrimaryKey.
getPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
                PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder
.createPrimaryKeyBuilder();
                mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_
Y_NAME_1, mainCalledNumber.getValue());
                mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_
Y_NAME_2, callStartTime.getValue());
                PrimaryKey mainTablePK = mainTablePKBuilder.build(); //
You can specify primary keys for the base table.

                // You can query the base table.
                SingleRowQueryCriteria criteria = new SingleRowQ
ueryCriteria(TABLE_NAME, mainTablePK);
                criteria.addColumnsToGet(colName); // You can read the
Duration attribute value of the base table.
                // You can specify 1 to indicate the latest data version
will be read.
                criteria.setMaxVersions(1);
                GetRowResponse getRowResponse = client.getRow(new
GetRowRequest(criteria));
                Row mainTableRow = getRowResponse.getRow();

                System.out.println(mainTableRow);
            }

            // If the nextStartPrimaryKey value is not null, you can
continue to read data from the base table.
            if (getRangeResponse.getNextStartPrimaryKey() != null) {
                rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
            } else {
                break;
            }
        }
    }

```

```
}

```

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

The following index is created.

IndexOnBaseStation2:

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

You can query the `IndexOnBaseStation2` index:

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime,
                                           long endTime,
                                           String colName) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX2_NAME);

    // You can specify primary keys.
    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());

    // You can specify primary keys.
    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(endPrimary
KeyBuilder.build());

        // You can specify the attribute name 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" + strBaseStationNum + "from" + strStartTime + "to" +
strEndTime + "is listed as follows:");
        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, you can
continue to read data from the base table.
            if (getRangeResponse.getNextStartPrimaryKey() != null) {
                rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
            } else {
                break;
            }
        }
    }
}

```

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

- If you want to fetch the following values from a result set: total call duration, the average call duration, the maximum call duration, and the minimum call duration. This result set is a value of `BaseStationNumber003` with a `StartTime` value range from 1532574861 to 1532584054.

Compared to the last query, return is not required for each call duration. However, return is required for duration statistics. You can fetch results using the same method as the last query. Then you can perform `Duration` calculations to obtain the required result. In addition, you can execute SQL statements in SQL-on-OTS to obtain statistics. For more information about how to activate SQL-on-OTS, see [OLAP on Table Store: serverless SQL big data analysis on Data Lake Analytics](#). You can

use most MySQL syntax in SQL-on-OTS. Additionally, with SQL-on-OTS, you can easily process complicated calculations that are applicable to your business.

8.4 Java SDK for global secondary indexes

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

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

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType.STRING)); // You can specify a primary key for a base table.
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType.INTEGER)); // Set primary key for the base table
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COLUMN_NAME_1, DefinedColumnType.STRING)); // You can specify a pre-defined attribute for the base table.
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COLUMN_NAME_2, DefinedColumnType.INTEGER)); // You can specify a pre-defined attribute for the base table.
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COLUMN_NAME_3, DefinedColumnType.INTEGER)); // You can specify a pre-defined attribute for the base table.

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

    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);

    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); // You can create an index.
    indexMeta.addPrimaryKeyColumn(DEFINED_COLUMN_NAME_1); // You can specify DEFINED_COLUMN_NAME_1 of the base table as an index primary key
    .
    indexMeta.addDefinedColumn(DEFINED_COLUMN_NAME_2); // You can specify DEFINED_COLUMN_NAME_2 of the base table as an index primary key
    .
    indexMetas.add(indexMeta); // You can add the index to the base table.

    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMetas); // You can create the base table.

    client.createTable(request);
}
```

- You can create an index on a base table.

```
private static void createIndex(SyncClient client) {
```

```

IndexMeta indexMeta = new IndexMeta(INDEX_NAME); // Create index
meta.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_2); // Specify
    DEFINED_COL_NAME_2 as the first primary key column of the index
    table.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // Specify
    DEFINED_COL_NAME_2 as the second primary key column of the index
    table.
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME
    , indexMeta, true); // Add the index table to the source table,
    including stock data
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME,
    indexMeta, false); // Add the index table to the source table, not
    including stock data
    client.createIndex(request); // Create an index table.
}

```

**Note:**

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

- **You can delete an index.**

```

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

```

- **You can read data from an index.**

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

```

private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
    eryCriteria(INDEX_NAME); // You can specify the name of an index.

    // You can specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
    createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1,
    PrimaryKeyValue.INF_MIN); // You can specify the minimum value for
    an index primary key.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
    PrimaryKeyValue.INF_MIN); // You can specify the minimum value for a
    base table primary key.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
    PrimaryKeyValue.INF_MIN); // You can specify the minimum value for a
    base table primary key.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
    ryKeyBuilder.build());

    // You can specify the end primary key.
}

```

```

        PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
        endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1,
PrimaryKeyValue.INF_MAX); // You can specify the maximum value for
an index attribute.
        endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MAX); // You can specify the maximum value for a
base table primary key.
        endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.INF_MAX); // You can specify the maximum value for a
base table primary key.
        rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimary
KeyBuilder.build());

        rangeRowQueryCriteria.setMaxVersions(1);

        System.out.println("The results returned from an index are as
follows:");
        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, you can
continue to read data from the base table.
            if (getRangeResponse.getNextStartPrimaryKey() != null) {
                rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
            } else {
                break;
            }
        }
    }
}

```

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

```

private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX_NAME); // You can specify the index name.

    // You can specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1,
PrimaryKeyValue.INF_MIN); // You can specify the minimum value for
an indexed attribute of an index.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MIN); // You can specify the minimum value for a
primary key of a base table.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.INF_MIN); // You can specify the minimum value for a
primary key of a base table.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
ryKeyBuilder.build());

    // You can specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();

```



```

        endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1,
        PrimaryKeyValue.INF_MAX); // You can specify the maximum value for
        an indexed attribute of an index.
        endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
        PrimaryKeyValue.INF_MAX); // You can specify the maximum value for a
        base table primary key.
        endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
        PrimaryKeyValue.INF_MAX); // You can specify the maximum value for a
        base table primary key.
        rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimary
        KeyBuilder.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.getPrimary
                KeyColumn(PRIMARY_KEY_NAME1);
                PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimary
                KeyColumn(PRIMARY_KEY_NAME2);
                PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder
                .createPrimaryKeyBuilder();
                mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1
                , pk1.getValue());
                mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2
                , ke2.getValue());
                PrimaryKey mainTablePK = mainTablePKBuilder.build(); //
                You can specify the index primary keys for a base table.

                // You can query a base table.
                SingleRowQueryCriteria criteria = new SingleRowQ
                ueryCriteria(TABLE_NAME, mainTablePK);
                criteria.addColumnsToGet(DEFINED_COL_NAME3); // You can
                read the DEFINED_COL_NAME3 attribute from the base table.
                // You can retrieve the latest data version.
                criteria.setMaxVersions(1);
                GetRowResponse getRowResponse = client.getRow(new
                GetRowRequest(criteria));
                Row mainTableRow = getRowResponse.getRow();
                System.out.println(row);
            }

            // If the value of nextStartPrimaryKey is not null, you can
            continue to read data from the base table.
            if (getRangeResponse.getNextStartPrimaryKey() != null) {
                rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
                getRangeResponse.getNextStartPrimaryKey());
            } else {
                break;
            }
        }
    }

```

```
}
```

8.5 APIs

CreateTable

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

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

CreateIndex

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



Note:

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

DeleteIndex

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

DeleteTable

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

8.6 Appendix

You can create tables and indexes as follows:

```
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 = "BaseStatio
nNumber";

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_COLUMN_NAME_1, DefinedColumnType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COLUMN_NAME_2, DefinedColumnType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COLUMN_NAME_3, DefinedColumnType.INTEGER));

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

        TableOptions tableOptions = new TableOptions(timeToLive,
        maxVersions);

        ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
        IndexMeta indexMeta0 = new IndexMeta(INDEX0_NAME);
        indexMeta0.addPrimaryKeyColumn(DEFINED_COLUMN_NAME_1);
        indexMetas.add(indexMeta0);
        IndexMeta indexMeta1 = new IndexMeta(INDEX1_NAME);
        indexMeta1.addPrimaryKeyColumn(DEFINED_COLUMN_NAME_3);
        indexMeta1.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMetas.add(indexMeta1);
        IndexMeta indexMeta2 = new IndexMeta(INDEX2_NAME);
        indexMeta2.addPrimaryKeyColumn(DEFINED_COLUMN_NAME_3);
        indexMeta2.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMeta2.addDefinedColumn(DEFINED_COLUMN_NAME_2);
        indexMetas.add(indexMeta2);

        CreateTableRequest request = new CreateTableRequest(tableMeta
        , tableOptions, indexMetas);

        client.createTable(request);
    }
```

9 Tunnel service

9.1 Overview

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

Background

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

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

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

Features

The following table lists the features provided by Tunnel Service.

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

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

9.2 Features

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

Tunnels for full and incremental data consumption

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

Orderly incremental data consumption

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

Consumption latency monitoring

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

Horizontal scaling of data consumption capabilities

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

9.3 Description of the data consumption framework

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

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

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

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

Automatic data processing

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

1. Initialize resources of the Tunnel client.

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

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

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

2. Regularly check heartbeat messages.

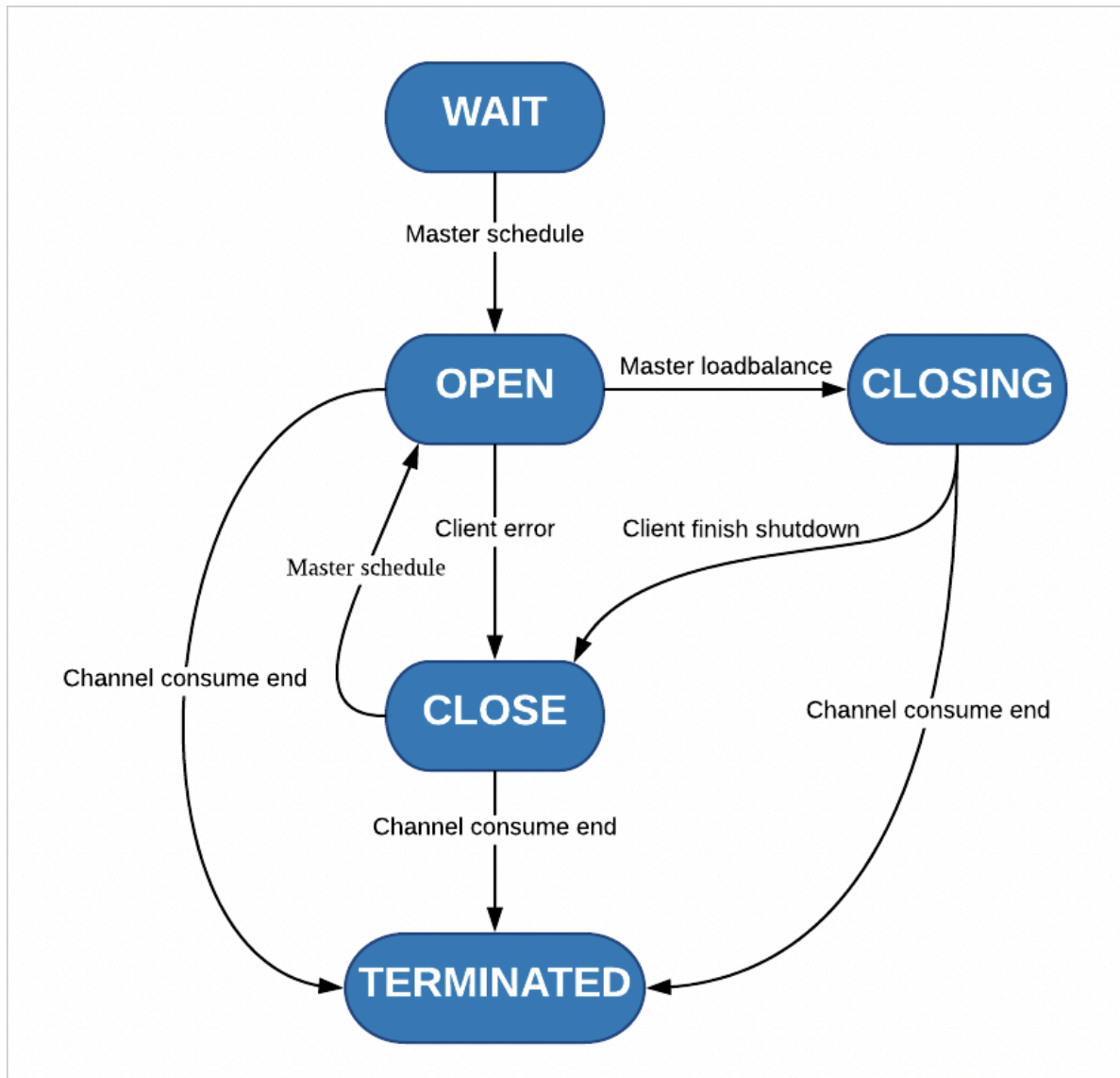
You can set the heartbeatIntervalInSec parameter in TunnelWorkerConfig to set the interval for checking the heartbeat.

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

3. Automatically process channel status.

Based on the number of active Tunnel clients obtained in the heartbeat request, Tunnel Service allocates available partitions to different clients to balance the

loads. Tunnel Service automatically processes channel status as described in the following figure, and drives channel consumption and load balancing.



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

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

Automatic load balancing and excellent horizontal scaling

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

Automatic resource clearing and fault tolerance

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

9.4 Quick start

You can use Tunnel Service in the Table Store console.

Prerequisites

You have [activated Table Store](#).

Create a tunnel

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

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

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

Preview data types in a channel

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

Enable data consumption for a tunnel

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

```
// Customize the data consumption callback, that is, implement the
// process and shutdown methods of the IChannelProcessor interface.
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 {
            // Mock record processing.
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}

// TunnelWorkerConfig contains more advanced parameters. For more
// information, see the description in the related topic.
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProc
essor());
// 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

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

9.5 SDKs

You can use the following SDKs to implement Tunnel Service:

- [Go SDK](#)
- [Java SDK](#)

9.6 Incremental synchronization performance white paper

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

Test environment

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

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

Test tools

- **Stress testing tool**

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

- **Pre-splitting tool**

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

- **Rate statistics tool**

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

Example

```
private static final Gson GSON = new Gson();
private static final int CAL_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.currentTimeMillis() - 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, it synchronizes data sequentially within a single channel to maintain the order of data, and synchronizes data in different channels in parallel. For incremental data, the number of channels is equal to the number of partitions in a table. This performance test focuses on how the number of partitions (channels) affects the incremental synchronization rate because the overall performance of Tunnel Service is greatly correlated with the number of partitions.

- **Test scenarios**

The test is conducted in the following scenarios:

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



Note:

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

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

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

Test indicators

This test uses the following indicators:

- **QPS (row):** the number of rows synchronized per second.
- **Average latency (ms per 1,000 rows):** the time required to synchronize 1,000 rows , in milliseconds.
- **CPU (core):** the total number of single-core CPUs used for data synchronization.
- **Memory (GB):** the total physical memory used for data synchronization.
- **Bandwidth (Mbit/s):** the total bandwidth used for data synchronization.



Note:

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

Test results

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

- **QPS and latency**

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

In the single-server 64-partition synchronization scenario, the gigabit NIC works at its full capacity, resulting in only 570,000 QPS. For more information, see

test details. The QPS in the double-server 64-partition synchronization scenario reaches 780,000, which is approximately twice the 420,000 QPS in the single-server 32-partition synchronization scenario. In the double-server 128-partition synchronization scenario, the QPS reaches 1,000,000.

- **System resource consumption**

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

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

- **Total bandwidth consumption**

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

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

Test details

- **Single-server single-channel: 19,000 QPS.**
 - Tested at: 17:40, January 30, 2019.
 - QPS: steady at approximately 19,000 rows per second, with a peak rate of 21,800 rows per second.
 - Latency: approximately 50 ms per 1,000 rows.
 - CPU usage: approximately 25% of a single-core CPU.
 - Memory usage: approximately 0.4% of the total physical memory, which is approximately 0.256 GB. (Each test server provides 64 GB physical memory.)
 - Bandwidth consumption: approximately 4,000 Kbit/s.
- **Single-server 4-partition synchronization: 70,000 QPS.**
 - Tested at: 20:00, January 30, 2019.
 - QPS: steady at approximately 70,000 rows per second, with a peak rate of 72,400 rows per second.
 - Latency: approximately 14.28 ms per 1,000 rows.
 - CPU usage: approximately 70% of a single-core CPU.
 - Memory usage: approximately 1.9% of the total physical memory, which is approximately 1.1 GB. (Each test server provides 64 GB physical memory.)
 - Bandwidth consumption: approximately 13 Mbit/s.
- **Single-server 8-partition synchronization: 130,000 QPS.**
 - Tested at: 20:20, January 30, 2019.
 - QPS: steady at approximately 130,000 rows per second, with a peak rate of 141,644 rows per second.
 - Latency: approximately 7.69 ms per 1,000 rows.
 - CPU usage: approximately 120% of a single-core CPU.
 - Memory usage: approximately 4.1% of the total physical memory, which is approximately 2.62 GB. (Each test server provides 64 GB physical memory.)
 - Bandwidth consumption: approximately 27 Mbit/s.

- **Single-server 32-partition synchronization: 420,000 QPS.**
 - Tested at: 15:50, January 31, 2019.
 - QPS: steady at approximately 420,000 rows per second, with a peak rate of 447,600 rows per second.
 - Latency: 2.38 ms per 1,000 rows.
 - CPU usage: approximately 450% of a single-core CPU.
 - Memory usage: approximately 8.2% of the total physical memory, which is approximately 5.25 GB. (Each test server provides 64 GB physical memory.)
 - Bandwidth consumption: approximately 86 Mbit/s.
- **Single-server 64-partition synchronization: 570,000 QPS, with the gigabit NIC working at its full capacity.**
 - Tested at: 22:10, January 31, 2019.
 - QPS: steady at approximately 570,000 rows per second, with a peak rate of 581,400 rows per second.
 - Latency: approximately 1.75 ms per 1,000 rows.
 - CPU usage: approximately 640% of a single-core CPU.
 - Memory usage: approximately 8.4% of the total physical memory, which is approximately 5.376 GB. (Each test server provides 64 GB physical memory.)
 - Bandwidth consumption: approximately 125 Mbit/s, which is the maximum rate of the gigabit NIC.
- **Double-server 64-partition synchronization: 780,000 QPS.**
 - Tested at: 22:30, January 31, 2019.
 - QPS: steady at approximately 390,000 rows per second on each server and 780,000 rows per second on both servers.
 - Latency: approximately 1.28 ms per 1,000 rows.
 - CPU usage: approximately 420% of a single-core CPU on each server and 840% of a single-core CPU on both servers.
 - Memory usage: approximately 8.2% of the total physical memory, which is approximately 10.5 GB. (Each test server provides 64 GB physical memory.)
 - Bandwidth consumption: approximately 169 Mbit/s. This indicates that bandwidth becomes the bottleneck when the number of partitions reaches 64 in single-server scenarios.

- **Double-server 128-partition synchronization: 1,000,000 QPS, with both gigabit NICs almost working at their full capacities.**
 - **Tested at: 23:20, January 31, 2019.**
 - **QPS: steady at approximately 500,000 rows per second on each server and 1,000,000 rows per second on both servers.**
 - **Latency: approximately 1 ms per 1,000 rows.**
 - **CPU usage: approximately 560% of a single-core CPU on each server and 1,020 % of a single-core CPU on both servers.**
 - **Memory usage: approximately 8.2% of the total physical memory, which is approximately 10.5 GB. (Each test server provides 64 GB physical memory.)**
 - **Bandwidth consumption: approximately 220 Mbit/s.**

Summary

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

10 HBase

10.1 Table Store HBase Client

In addition to SDKs and RESTful APIs, Table Store HBase Client can be used to access Table Store through Java applications built on open source HBase APIs.

Based on Java SDKs for Table Store version 4.2.x and later, Table Store HBase Client supports open source APIs for HBase version 1.x.x and later.

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

- [GitHub tablestore-hbase-client project](#)
- [Compressed package](#)
- **Maven**

```
<dependencies>
  <dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>tablestore-hbase-client</artifactId>
    <version>1.2.0</version>
  </dependency>
</dependencies>
```

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

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

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

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

10.2 Table Store HBase Client supported functions

API support differences between Table Store and HBase

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

Functions supported by Table Store HBase Client APIs:

- **CreateTable**

Table Store does not support ColumnFamily as all data can be considered to be in the same ColumnFamily. This means that TTL and Max Versions of Table Store

are at the table-level. Therefore, Table Store has some support for the following functions:

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

- Put

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

- **Get**

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

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

- **Scan**

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

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

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

- Batch

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

- Delete

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

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

- **checkAndXXX**

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

- **Exist**

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

- **Filter**

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

Functions not supported by Table Store HBase Client APIs

- **Namespaces**

Table Store uses instances to manage a data table. An instance is the minimum billing unit in Table Store. You can manage instances in the [Table Store console](#).

Therefore, the following features are not supported:

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

- **Region management**

[Data partition](#) is the basic unit for data storage and management in Table Store.

Table Store automatically splits or merges the data partitions based on their data volumes and access conditions. Therefore, Table Store does not support features related to Region management in HBase.

- **Snapshots**

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

- **Table management**

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

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

- **Coprocessors**

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

- `coprocessorService()`
- `coprocessorService(ServerName serverName)`
- `getMasterCoprocessors()`

- **Distributed procedures**

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

- `execProcedure(String signature, String instance, Map props)`
- `execProcedureWithRet(String signature, String instance, Map props)`
- `isProcedureFinished(String signature, String instance, Map props)`

- **Increment and Append**

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

10.3 Differences between Table Store and HBase

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

Table

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

Row and Cell

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

GET

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

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

SCAN

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

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

Batch

Table Store does not support BatchCallback.

Mutations and Deletions

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

Increment and Append

Table Store does not support Increment or Append features.

Filter

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

Optimization

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

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

Admin

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

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

- **CreateTable**

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

- **Maintenance task**

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

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

- **Namespaces**

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

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

- **Region**

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

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

Snapshots

Table Store does not support Snapshots related APIs.

Replication

Table Store does not support Replication related APIs.

Coprocessors

Table Store does not support Coprocessors related APIs.

Distributed procedures

Table Store does not support Distributed procedures related APIs.

Table Management

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

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

Restrictions

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

10.4 Migrate from HBase to Table Store

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

Dependencies

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

```
<dependencies>
  <dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>tablestore-hbase-client</artifactId>
    <version>1.2.0</version>
  </dependency>
```

```
</dependencies>
```

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

```
<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>
```

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

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

If you want to use HBase Client version v0.x.x, see [Migrate HBase of an earlier version](#).

Configure the file

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

- HBase Connection type

Set Connection to TableStoreConnection.

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



```
</property>
```

- **Configuration items of Table Store**

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

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

```
<property>
  <name>tablestore.client.endpoint</name>
  <value></value>
</property>
<property>
  <name>tablestore.client.instanceName</name>
  <value></value>
</property>
<property>
  <name>tablestore.client.accessKeyId</name>
  <value></value>
</property>
<property>
  <name>tablestore.client.accessKeySecret</name>
  <value></value>
</property>
```

- **Optional items you can configure are as follows.**

```
<property>
  <name>hbase.client.tablestore.family</name>
  <value>f1</value>
</property>
<property>
  <name>hbase.client.tablestore.family.$tablename</name>
  <value>f2</value>
</property>
<property>
  <name>tablestore.client.max.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**

- **Table Store only supports single ColumnFamilies. When you use HBase APIs, you must enter the content of the family.**

`hbase.client.tablestore.family` indicates global configuration, while `hbase.client.tablestore.family.$tablename` indicates configuration of a single table.

- **Rule: For tables whose names are T, search for `hbase.client.tablestore.family.T` first. If the family does not exist, search for `hbase.client.tablestore.family`. If the family does not exist, use the default value f.**

- **tablestore.client.max.connections**

Maximum connections. The default value is 300.

- **tablestore.client.socket.timeout**

Socket time-out time. The default value is 15 seconds.

- **tablestore.client.connection.timeout**

Connection time-out time. The default value is 15 seconds.

- **tablestore.client.operation.timeout**

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

- **tablestore.client.retries**

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

10.5 Migrate HBase of an earlier version

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

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

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

Connection APIs

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

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

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

The latest code is as follows:

```
Connection connection = ConnectionFactory.createConnection(config);
// ...
connection.close();
```

TableName series

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

The latest code is as follows:

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

Table, BufferedMutator, and RegionLocator APIs

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

- `org.apache.hadoop.hbase.client.Table`: **Used to operate reading, writing, and other requests of a single table.**
- `org.apache.hadoop.hbase.client.BufferedMutator`: **Used for asynchronous batch writing. This API corresponds to `setAutoFlush(boolean)` of the `HTableInterface` API of the earlier versions.**
- `org.apache.hadoop.hbase.client.RegionLocator`: **Indicates the table partition information.**

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

Admin APIs

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

Use the Connection instance to create an Admin instance:

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

10.6 Hello World

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

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

Code position

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

Use HBase APIs

- Configure project dependencies

Configure Maven dependencies as follows.

```
<dependencies>
  <dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>tablestore-hbase-client</artifactId>
```

```
<version>1.2.0</version>
</dependency>
</dependencies>
```

For more information about advanced configurations, see [Migrate from HBase to Table Store](#).

- **Configure the file**

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

```
<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 advanced configurations, see [Migrate from HBase to Table Store](#).

- **Connect Table Store**

Create a `TableStoreConnection` object to connect Table Store.

```
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();
```

- **Create a table**

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

```
// Create an HTableDescriptor, which contains only one ColumnFamily
HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf(TABLE_NAME));

// Create a ColumnFamily. Use the default ColumnFamily name for Max Versions and TimeToLive. The default ColumnFamily name for Max Versions is 1 and for TimeToLive is Integer.INF_MAX
descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));

// Use the createTable API of the Admin to create a table
System.out.println("Create table " + descriptor.getNameAsString());
admin.createTable(descriptor);
```

- **Write Data**

Write a row of data to Table Store.

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

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

// Add a column. Table Store supports only single ColumnFamilies. The ColumnFamily name is configured in hbase-site.xml. If the ColumnFamily name is not configured, the default name is "f". In this case, the value of COLUMN_FAMILY_NAME may be null when data is written.
put.addColumn(COLUMN_FAMILY_NAME, COLUMN_NAME, COLUMN_VALUE);

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

- **Read Data**

Read data of the specified row.

```
// Create a Get object to read the row whose primary key is ROW_KEY.
Result getResult = table.get(new Get(ROW_KEY));
Result result = table.get(get);

// Print the results
String value = Bytes.toString(getResult.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME));
System.out.println("Get one row by row key");
```

```
System.out.printf("\t%s = %s\n", Bytes.toString(ROW_KEY), value);
```

- **Scan data**

Read data in the specified range.

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

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));
}
```

- **Delete a table**

Use Admin APIs 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("HelloTable
store");
    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_NAME));
        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 HelloTable
store: " + e.toString());
        System.exit(1);
    }
}

```


11 Authorization management

11.1 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 Table Store, 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.

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 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 Table Store. 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.
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 Table Store instances, all Table Store 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 Table Store 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.

The following topics provide examples to show how to use RAM and STS and suggestions on their usage. Operations in the examples are performed through the console and command lines to reduce the actual usage of code. If code must be used to perform such operations, see the [API Reference \(RAM\)](#) and [API Reference \(STS\)](#).

11.2 Create a RAM user account

This topic describes how to create a RAM user account.

Step 1: Log on to the RAM console

You can use your Alibaba Cloud account to log on to the RAM console to create a RAM user account. If your Alibaba Cloud account grants a RAM user administrative permissions, the RAM user will also be able to create multiple RAM user accounts through the RAM console.

- Log on to the [RAM console](#) with an Alibaba Cloud account.
- Log on to the [RAM console](#) with a RAM user account.

Step 2: Create a RAM user account

Procedure

1. In the left-side navigation pane, click Users under Identities.
2. Click Create User.



Note:

To create multiple RAM users at a time, click Add User.

3. Specify the Logon Name and Display Name parameters.
4. Under Access Mode, select Console Password Logon or Programmatic Access.
 - **Console Password Logon:** If you select this check box, you must also complete the basic security settings for logon, including deciding whether to automatically generate a password or customize the logon password, whether

the user must reset the password upon the next logon, and whether to enable multi-factor authentication (MFA).

- **Programmatic Access:** If you select this check box, an AccessKey pair is automatically created for the RAM user. The user can access Alibaba Cloud resources by calling an API operation or by using a development tool.



Note:

We recommend that you select only one access mode for the RAM users to ensure the security of your Alibaba Cloud account. This prevents RAM users who have terminated their employment contracts with the company from accessing Alibaba Cloud resources.

5. Click OK.

11.3 Grant permissions to a RAM user

This topic describes how to grant permissions to a RAM user.

Table Store permission policies

RAM provides three permission policies for Table Store:

- **AliyunOTSReadOnlyAccess** (read-only permissions on Table Store)
- **AliyunOTSWriteOnlyAccess** (write-only permissions on Table Store)
- **AliyunOTSFullAccess** (management permissions on Table Store)

You can choose from the preceding policies to grant permissions to RAM users as needed. You can also [#unique_111](#).

Procedure

Procedure

1. Log on to the [RAM console](#) by using your Alibaba Cloud account.
2. In the left-side navigation pane, choose Permissions > Grants.
3. Click Grant Permission.
4. In the Principal field, enter the principal name and click the target principal.



Note:

You can enter a keyword to search for a RAM user, user group, or role.

5. In the Policy Name column, select the target policy.



Note:

You can click X to revoke your selection.

6. Click OK.

7. Click Finished.

11.4 Configure an MFA device for a RAM user

This topic describes how to use Multi-Factor Authentication (MFA) to enhance security for your account.

Context

Context

MFA is a simple and effective authentication method that adds an extra layer of security in addition to username and password. After MFA is configured, when a RAM user logs on to the Alibaba Cloud website, the system requires the user to enter the username and password (first security factor), and then requires the user to enter a dynamic verification code (second security factor) from the MFA device. The multi-factor authentication provides greater security for your account.

Procedure

The following section uses Google Authenticator app as an example to describe how to configure an MFA device for a RAM user.

Procedure

1. In the left-side navigation pane, click Users under Identities.
2. In the User Logon Name/Display Name column, click the username of the target RAM user.
3. On the Authentication tab, click Enable the virtual MFA device.
4. Download and install the Google Authenticator app on your mobile phone.
 - For iOS, install the Google Authenticator app from the App Store.
 - For Android, install the Google Authenticator app from the Google Play Store.



Note:

You need to install a QR code scanner from the Google Play Store for Google Authenticator to identify QR codes.

5. Open the Google Authenticator app and tap BEGIN SETUP.

6. Select a method to enable the MFA device from the following available options.

- **(Recommended) Tap Scan barcode in the Google Authenticator app and scan the QR code displayed on the Scan the code tab in the RAM console.**
- **Tap Manual entry, enter the username and key, and then tap the ✓ icon in the Google Authenticator app.**



Note:

You can obtain the username and key from the Retrieve manually enter information tab in the RAM console.

7. Enter the two consecutive verification codes that are obtained from the Google Authenticator app, and click Enable.



Note:

The verification code in the Google Authenticator app is refreshed at an interval of 30 seconds.

What to do next

When a RAM user logs on to the RAM console with the MFA device enabled, the RAM user must enter the following information:

- 1. Username and password of the RAM user**
- 2. Two consecutive verification codes provided by the MFA device**



Note:

Before you uninstall or remove an MFA device, you must log on to the Alibaba Cloud console and disable the MFA device. Otherwise, a logon failure may occur.

11.5 STS temporary access authorization

11.5.1 Create a temporary role and grant permissions

Security Token Service (STS) is a permission management system provided by Alibaba Cloud. You can use policies specified through STS to control user

permissions. This topic describes how to use STS to authorize users temporary permissions to access Table Store.

Context

Context

In typical app development scenarios, you can use STS to set temporary access permissions for different users. You can specify the validity period of the temporary token to mitigate the risks of RAM user account information being leaked. Different authorization policies can be added to control the access permissions of different app users. For example, you can control the table paths accessed by users to isolate the storage spaces of different app users.

Prerequisites

- Log on to the [RAM console](#) with an Alibaba Cloud account.
- You have created a RAM user account named `ram_test_app`. When a RAM user account assumes a role, the account automatically obtains all role permissions and does not need to be granted further permissions.

Step 1: Create temporary roles

Create two roles: `RamTestAppReadOnly` and `RamTestAppWrite`. Grant read permissions to `RamTestAppReadOnly` and file upload permissions to `RamTestAppWrite`. Perform the following operations:

Procedure

1. In the left-side navigation pane, click RAM Roles.
2. Click Create RAM Role, select Alibaba Cloud Account, and then click Next.
3. Specify the RAM Role Name and Note parameters.
4. Under Select Trusted Alibaba Cloud Account, select Current Alibaba Cloud Account or Other Alibaba Cloud Account.



Note:

If you select Other Alibaba Cloud Account, enter the account ID.

5. Click OK.

Step 2: Create custom policies

Repeat the following steps to create two policies named `ram-test-app-readonly` and `ram-test-app-write`.

Procedure

1. In the left-side navigation pane, click Policies under Permissions.
2. On the page that appears, click Create Policy.
3. On the Create Custom Policy page, specify the Policy Name and Note parameters.
4. Under Configuration Mode, select Visualized or Script.
 - If you select Visualized, click Add Statement. On the page that appears, configure the permission effect, actions, and resources.
 - If you select Script, edit the policy script according to the [policy structure and syntax](#).
5. Click OK.

The scripts for the ram-test-app-readonly policy and ram-test-app-write policy in this example are as follows:

- 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"
}
```

Step 3: Assign policies to roles

Repeat the following steps to assign the ram-test-app-readonly (read-only permissions on Table Store) policy to RamTestAppReadOnly and assign the ram-test-app-write (write-only permissions on Table Store) policy to RamTestAppWrite.

Procedure

1. In the left-side navigation pane, click Grants under Permissions.
2. Click Grant Permission.
3. Under Principle, enter the RAM role name, and click the target RAM role.
4. In the Policy Name column, select the target policies by clicking the corresponding rows.



Note:

You can click X in the section on the right side of the page to delete the selected policy.

5. Click OK.
6. Click Finished.

Subsequent operations

[Authorize temporary access](#)

11.5.2 Authorize temporary access

After creating roles, you can use STS to grant RAM users temporary permissions to access Table Store.

Prerequisites

You have completed the operations described in [Create a temporary role and grant permissions](#).

Step 1: Authorize a RAM user account to assume roles

Before using STS to authorize access, you must authorize the RAM user account to assume roles. Unpredictable risks may occur if any RAM user account could assume these roles. Therefore, a RAM user account must have explicitly configured permissions to assume the corresponding role. To create two custom authorization policies and assign them to the RAM user account ram_test_app, perform the following steps:

Procedure

1. Create two custom authorization policies:



Note:

For more information about how to create custom authorization policies, see [#unique_111](#).

• 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"
    }
  ]
}
```

2. Assign the two policies to the RAM user account ram_test_app.



Note:

For more information, see [#unique_117](#).


Step 2: Use STS for authorized access

After authorizing roles for a RAM user account, you can use STS for authorized access. You must download the required Python command line tool of STS from [sts.py](#).

The call method is as follows. For more information about the parameters, see [API References \(STS\)](#).

```
$python ./sts.py AssumeRole RoleArn=acs:ram::198***237:role/ramtestapp
readonly RoleSessionName=usr001 Policy='{ "Version": "1", "Statement
": [{" "Effect": "Allow", "Action": ["ots:ListTable", "ots:DescribeTable
"], "Resource": ["acs:ots:*:*:ram-test-app", "acs:ots:*:*:ram-test-app/
*"] } ] }' DurationSeconds=1000 --id=id --secret=secret
```

Parameters:

Parameter	Description
RoleArn	The ID of the role to be assumed. You can log on to the RAM console. On the Role Management page, click Manage in the Actions column corresponding to a role name. On the Basic Information page of the role, find the Arn.
RoleSessionName	The name of a temporary credential. We recommend that you use different application users to distinguish credentials.
Policy	<p>The permissions added when a role is assumed.</p> <div> Note: The added policy is used to control the permissions of the temporary credential after the role is assumed. The permissions obtained by the temporary credential are restricted by both the role and the added policy. When a role is assumed, a policy can be added to further control the permissions. For example, when uploading files, you can add a policy to control upload paths for different users.</div>

Parameter	Description
DurationSeconds	The validity period of the temporary credential. Unit: seconds. Valid values: 900 to 3600.
id and secret	The AccessKey ID and AccessKey secret of the RAM user account to assume a role.

Test STS

Create a table named `test_write_read` and specify the `name` column as the primary key and of the string type in the [Table Store console](#). Then, use the CLI tool to test the read and write operations on Table Store.

Use the RAM user account `ram_test_app` to access Table Store. Replace the AccessKey pair in the following example with your own AccessKey pair for testing.

```
python2.7 ots_console --url https://TableStoreTest.cn-hangzhou.ots.aliyuncs.com --id <yourAccessKeyId> --key <yourAccessKeySecret>
You cannot access the instance!
ErrorCode: OTSNoPermissionAccess
ErrorMessage: You have no permission to access the requested resource
, please contact the resource owner.
```

The access failed because the RAM user account `ram_test_app` does not have permissions to access the resources.

Use temporary permissions to read and write data as well as accessing the console

- Use the temporary permission to write data

Use STS to write data. In this example, the added policy is the same as that of the role. The default value 3600 of `DurationSeconds` is used, and `SessionName` is set to `session001`. Perform the following steps:

1. Use STS to obtain a temporary credential.

```
python2.7 ./sts.py AssumeRole RoleArn=acs:ram::198***237:role/ramtestappwrite RoleSessionName=session001 Policy='{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ots:Create*",
        "ots:BatchWrite*",
        "ots:Put*",
        "ots:Insert*",
        "ots:Update*",
        "ots:Delete*"
      ],
      "Resource": [
        "acs:ots:*:*:instance/ram-test-app",
        "acs:ots:*:*:instance/ram-test-app/table/*"
      ]
    }
  ],
  "Version": "1"
}' --id=<yourAccessKeyId> --secret=<yourAccessKeySecret>
{
  "AssumedRoleUser": {
    "Arn": "acs:ram::198***237:role/ramtestappwrite/session001",
    "AssumedRoleId": "33062905274959****:session001"
  },
  "Credentials": {
```

```

    "AccessKeyId": "****",
    "AccessKeySecret": "****"
    "SecurityToken": "CAE****0ZQ=="
  },
  "RequestId": "5F92B248-F200-40F8-A05A-C9C7D018E351"
}

```

2. Use the CLI tool to write data. The token parameter will be supported in the upcoming V1.2.

```

python2.7 ots_console --url https://TableStoreTest.cn-hangzhou
.ots.aliyuncs.com --id <yourAccessKeyId> --key <yourAccess
KeySecret> --token=CAE****0ZQ==

OTS-TableStoreTest>$ put test_write_read '001' age:integer=30
A new row has been put in table test_write_read

```

- Use the temporary permission to read data

Use STS to read data. In this example, the added policy is the same as that of the role. The default value 3600 of DurationSeconds is used, and SessionName is set to session002. Perform the following steps:

1. Use STS to obtain a temporary credential.

```

python2.7 ./sts.py AssumeRole RoleArn=acs:ram::198***237:role/
ramtestappreadonly RoleSessionName=session002 Policy='{
  "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"
}' --id=6iT***lRt --secret=****
{
  "AssumedRoleUser": {
    "Arn": "acs:ram::198***237:role/ramtestappreadonly/session002",
    "AssumedRoleId": "396025752746614078:session002"
  },
  "Credentials": {
    "AccessKeyId": "****",
    "AccessKeySecret": "****",
    "Expiration": "2017-06-09T09:17:19Z",
    "SecurityToken": "CAE****seQ=="
  },
  "RequestId": "EE788165-B760-4014-952C-E58ED229C80D"
}

```

2. Use the CLI tool to read data. The token parameter will be supported in the upcoming V1.2.

```

python2.7 ots_console --url https://TableStoreTest.cn-hangzhou
.ots.aliyuncs.com --id STS***Q8Q --key **** --token=CAE****Q==

OTS-TableStoreTest>: get test_write_read '001'
age:INTEGER='30'

```

- Use the temporary permission to access the console

STS temporary authorization allows you to use RAM user accounts to log on to the Table Store console and manage and view instances and table resources under the Alibaba Cloud account. In the preceding example, the RAM user account `ram_test_app` can assume the role `RamTestAppReadOnly` and have the corresponding permissions to view all instances and tables. Perform the following steps to log on to the console:

1. Log on to the [RAM console](#) with an Alibaba Cloud account.
2. Log on to the RAM console with an Alibaba Cloud account and go to the Overview page.
3. Click the link below Account Management. On the RAM User Logon page, set RAM User Name and Password.
4. After you log on to the console, move the pointer over the username in the upper-right corner. In the message that appears, click Switch Role.
5. On the Switch Role page that appears, enter an enterprise alias and a role name to which you want to switch. Click Switch.

Step 4: Use the temporary permission to call the JAVA SDK

Create an `OTSClient` object and add the `AccessKeyId`, `AccessKeySecret`, and `Token` parameters of the STS Token as shown in the following example:

```
OTSClient client = new OTSClient(otsEndpoint, stsAccessKeyId,
    stsAccessKeySecret, instanceName, stsToken);
```

11.6 Custom permissions

This topic describes the definitions and application scenarios of Action, Resource, and Condition.

Action

Action defines the specific API operation or operations to allow or deny. When creating a Table Store authorization policy, add the `ots:` prefix to each API operation and separate different API operations with commas (,). The asterisk (*) wildcard is used in Action to specify the prefix matching and suffix matching.

Action is defined as follows:

- **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:*"
```

Resource

Resource in Table Store is composed of multiple fields, including the service, region, user ID, instance name, and table name. Each field supports asterisk (*) wildcards for prefix and suffix matching. The format is as follows:

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

The fields enclosed in brackets are variables. The value of the region field must be region IDs, such as cn-hangzhou. The user_id field is set to an Alibaba Cloud account ID.



Note:

- **Table Store instance names are not case-sensitive. However, the `instance_name` field in Resource must be in lower case.**
- **Resource is defined for Tunnel Service by instances rather than tables and includes fields such as service, region, user ID, and instance name in the definition. The format is as follows:**

```
acs:ots:[region]:[user_id]:instance/[instance_name]
```

Resource is defined as follows:

- **All resources of users in all regions**

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

- **All instances and their tables of User 123456 in China (Hangzhou)**

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

- **Instance abc and its tables of User 123456 in China (Hangzhou)**

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

- **All instances with the prefix abc and their tables**

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

- **All instances with the prefix abc and their tables with the prefix xyz. Instance resources do not match `acs:ots:*:*:instance/abc*`.**

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

- **All instances with the suffix abc and their tables with the suffix xyz**

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

Table Store API operations

Table Store provides two types of API operations:

- **Management API operations for reading from and writing to instances.**

- Data API operations for reading from and writing to tables and rows.

Details about these API operations are as follows:

- Resources for management API operations

Management API operations are instance-based operations and can only be called through the console. Specifying Action and Resource for Management API operations determines subsequent use of the console. The `acs:ots:[region]:[user_id]:` prefix is omitted in the following accessed resources. Only the instance and table are described.

API operation/Action	Resource
ListInstance	instance/*
InsertInstance	instance/[instance_name]
GetInstance	instance/[instance_name]
DeleteInstance	instance/[instance_name]

- Resources for data API operations

Data API operations are table- and row-based operations, which can be called through the console or by the SDK. Specifying Action and Resource for data API operations determines subsequent use of the console. The `acs:ots:[region]:[user_id]:` prefix is omitted in the following accessed 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]

API operation/Action	Resource
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]
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]

API operation/Action	Resource
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]
CosumeTunnel	instance/[instance_name]/table/[table_name]

- Resources for Tunnel Service API operations

API operations for Tunnel Service are instance-based operations and can be called through the console or by the SDK. Specifying Action and Resource for Tunnel Service API operations determines subsequent use of the console. The `acs:ots:[region]:[user_id]: prefix` is omitted in the following accessed 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]
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]

API operation/Action	Resource
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]
CosumeTunnel	instance/[instance_name]

- **Instructions**

- Action and Resource in a policy are verified by string matching. The asterisk (*) wildcard is used to specify the prefix matching and suffix matching. If Resource is defined as `acs:ots:*:*:instance/*`, `acs:ots:*:*:instance/abc` cannot be matched. If Resource is defined as `acs:ots:*:*:instance/abc`, `acs:ots:*:*:instance/abc/table/xyz` cannot be matched.
- To use a RAM user account to manage instance resources through the Table Store console, the RAM user account must be granted `read` permissions on `acs:ots:[region]:[user_id]:instance/*` because the console needs to obtain the instance list.
- For batch API operations, such as `BatchGetRow` and `BatchWriteRow`, the backend service authenticates each table to be accessed. Operations can only be performed when all tables are authenticated. Otherwise, an error message is returned.

Condition

Policies can support a variety of authentication conditions, including IP address-based access control, HTTPS-based access control, Multi-Factor Authentication (

MFA)-based access control, and time-based access control. These conditions are supported by all Table Store API operations.

- IP address-based access control

RAM allows you to specify IP addresses or CIDR blocks that are used to access Table Store resources. Typical application scenarios are as follows:

- Specify multiple IP addresses. For example, the following code indicates that only access requests from IP addresses 10.101.168.111 and 10.101.169.111 are allowed.

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "ots:*",
      "Resource": "acs:ots:*:*:*:*",
      "Condition": {
        "IpAddress": {
          "acs:SourceIp": [
            "10.101.168.111",
            "10.101.169.111"
          ]
        }
      }
    }
  ],
  "Version": "1"
}
```

- Specify one IP address or CIDR block. For example, the following code indicates that only access requests from IP address 10.101.168.111 or CIDR block 10.101.169.111/24 are allowed.

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "ots:*",
      "Resource": "acs:ots:*:*:*:*",
      "Condition": {
        "IpAddress": {
          "acs:SourceIp": [
            "10.101.168.111",
            "10.101.169.111/24"
          ]
        }
      }
    }
  ],
  "Version": "1"
}
```

- **HTTPS-based access control**

RAM allows you to specify whether resources must be accessed by requests over HTTPS.

The following example indicates that Table Store resources must be accessed by requests over HTTPS.

```
{
  "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 resources must be accessed by requests that have passed MFA.

The following example indicates that Table Store resources must be accessed by 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 are allowed to 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 specific policies in typical scenarios and offers authorization methods based on the definitions of Action, Resource, and Condition.

- **Multiple authorization conditions**

In this scenario, RAM users using the 10.101.168.111/24 CIDR block are allowed to read from and write to all instances named online-01 and online-02 (including

all tables of these instances). Access is only allowed before 0:00:00 January 1, 2016, and all access requests must be made over HTTPS.

The procedure is as follows:

1. Log on to the [RAM console](#) with an Alibaba Cloud account. (Assume that RAM is activated.)
2. In the left-side navigation pane, choose Permissions > Policies to go to the Policies page.
3. Click Create Policy to go to the Create Custom Policy page.
4. Enter Policy Name and select Script as the Configuration Mode. 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"
}
```

5. Click OK.
6. In the left-side navigation pane, choose Identities > Users. On the Users page that appears, click Add Permissions in the Actions column corresponding to a RAM user account.
7. In the Add Permissions dialog box that appears, search for the newly created policy, and click the policy to add the permissions to the Selected column. Click OK. The selected permissions are granted to the RAM user account.

- **Reject requests**

In this scenario, RAM users using the IP address 10.101.169.111 are not allowed to write to any tables that belong to instances prefixed with **online** or **product** and located in China (Beijing). This policy does not define actions and permissions on instances.

To reject requests, perform the steps described in the preceding "Multiple authorization conditions" section to create a new policy and grant policy permissions to the designated RAM user. Copy the following content to Policy Document during policy creation:

```
{
  "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"
}
```

11.7 Authorize a RAM user account to log on to the console

Table Store allows you to create and manage instances in the Table Store console.

You can use your Alibaba Cloud account to grant RAM user accounts permissions to log on to the Table Store console.

Procedure

To grant a RAM user account the permission to log on to the console, perform the following steps:

Procedure

1. Log on to the [RAM console](#) with an Alibaba Cloud account.
2. In the left-side navigation pane, click Users.
3. On the User Management page that appears, click Manage in the Actions column corresponding to a RAM user account to go to the User Details page.
4. In the Access Mode section, select Console Password Logon.
5. Set a password for the RAM user account to log on to the Alibaba Cloud console, and click OK.
6. In the left-side navigation pane, click Dashboard to go to the RAM Overview page.
7. Click the URL following RAM User Logon Link to go to the RAM User Logon page. Use the password of the RAM user account set in Step 5 to log on to the Alibaba Cloud console.
8. In the left-side navigation pane, click the Table Store icon to go to the Table Store console.

11.8 Examples

If you need to share the data of a Table Store instance under your Alibaba Cloud account to others but do not want the data to be modified, you can create a RAM user account and grant the read-only permission to the account. This example describes how to separate read and write permissions by granting the permissions to different RAM user accounts.

Create a RAM user account

Procedure

1. Log on to the [RAM console](#) with an Alibaba Cloud account.
2. In the left-side navigation pane, click Users to go to the User Management page.
3. In the upper-right corner of the page, click Create User to open the Create User dialog box.
4. Specify the required information, and select Automatically generate an Access key for this user. Click OK.

**Note:**

For this example, username ram_test is used.

5. After you create a RAM user account, an AccessKey pair is generated for the account. Click Save Access Key Information.

**Note:**

After an AccessKey pair is generated, you cannot view the AccessKey pair in the console. You must save your AccessKey pair and keep it confidential.

**Note:**

On the User Details page, you can also select Enable Console Logon for the RAM user account.

Grant permissions to a RAM user account

Procedure

1. On the User Management page, click ram_test to go to the User Details page of the RAM user account.
2. In the left-side navigation pane, click User Authorization Policies.
3. In the upper-right corner of the page, click Edit Authorization Policy.
4. In the dialog box that appears, search for Table Store permissions. The corresponding permissions are displayed on the left side of the dialog box.
5. Select permissions. Click > to add the permissions to the right section of the dialog box. Click OK.

**Note:**

For this example, grant AliyunOTS ReadOnlyAccess (read-only permission on Table Store) to ram_test.

**Note:**

On the User Details page, you can also select Enable Console Logon for the RAM user account.

Test example

Use the AccessKey pair of the created RAM user account to test whether the account has the permissions to create and delete tables. You must replace the AccessKey pair used in the following example with your own AccessKey pair.

```
$python ots_console --url https://TableStoreTest.cn-hangzhou.ots.aliyuncs.com --id <yourAccessKeyId> --key <yourAccessKeySecret>
```

```
$OTS-TableStoreTest>: ct test pk1:string,pk2:integer readrt:1 writert:1
Fail to create table test.

$OTS-TableStoreTest>: dt test
You will delete the table:test!

press Y (confirm) :Y
Fail to delete table test.
```

The RAM user account `ram_test` cannot create or delete tables because it has only been granted read permissions. You can follow the preceding steps to create a RAM user account with the read-only permission for your Alibaba Cloud account.