

Alibaba Cloud Table Store

Functions

Issue: 20190523

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

Generic conventions

Table -1: Style conventions

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

Style	Description	Example
<code>{}</code> or <code>{a b}</code>	It indicates that it is a required value, and only one item can be selected.	<code>swich {stand slave}</code>

Contents

Legal disclaimer.....	I
Generic conventions.....	I
1 Table Store tables.....	1
2 Conditional update.....	4
3 Auto-increment function of the primary key column.....	7
4 Description of the data consumption framework.....	10
5 Stream.....	14
5.1 Overview.....	14
5.2 Stream API/SDK.....	17
5.3 Stream Client.....	18
6 HBase.....	26
6.1 Table Store HBase Client.....	26
6.2 Table Store HBase Client supported functions.....	27
6.3 Differences between Table Store and HBase.....	33
6.4 Migrate from HBase to Table Store.....	38
6.5 Migrate HBase of an earlier version.....	41
6.6 Hello World.....	43
7 SearchIndex.....	49
7.1 Overview.....	49
7.2 Features.....	50
8 Global secondary indexes.....	53
8.1 Overview.....	53
8.2 Introduction.....	55
8.3 Scenarios.....	57
8.4 Java SDK for global secondary indexes.....	66
8.5 APIs.....	70
8.6 Billing rules.....	71
8.7 Appendix.....	78

1 Table Store tables

When creating a Table Store table, you must specify a table name, a primary key, and reserved read/write throughput.

Naming conventions

Table Store table names:

- Can contain uppercase letters, lowercase letters, digits, and underscores.
- Must start with an uppercase letter, an lowercase letter, or an underscore.
- Are case sensitive.
- Must be 1 to 255 characters in length.
- Must be unique within the same instance (tables in different instances are allowed to use the same name).

Primary Key

When creating a Table Store table, you must specify the primary key of the table. A primary key contains at least one, and up to four primary key columns. Each primary key column has a name and type. Table Store has some restrictions on the names and types of the primary key columns. For more information, see [Primary key and attribute](#).

Table Store indexes data based on the primary key. The primary key uniquely identifies each row in the table, so that no two rows have the same key. The rows are sorted in ascending order by their primary key.

Reserved read/write throughput

To guarantee the consistent and low-latency performance of Table Store, you can specify the reserved read/write throughput during table creation. If the value of the reserved read/write throughput is not 0, Table Store reserves the necessary capacity to meet the specified throughput requirements. At the same time, costs are determined based on the reserved read/write throughput. You can dynamically raise and lower the reserved read/write throughput based on business requirements. The reserved read/write throughput is set in quantities of read capacity units and write capacity units.



Note:

Tables created in capacity instances do not support the reserved read/write throughput.

You can update the tables reserved read/write throughput through the UpdateTable operation. The rules for updating the reserved read/write throughput are as follows.

- A time interval of at least two minutes is required between two updates for the same table. For example, if you update the reserved read/write throughput of a table at 12:43:00, you must wait until after 12:45:00 to update the table for a second time. The required 2-minute time interval between updates is applied at the table level. Between 12:43:00 and 12:45:00, you can update the reserved read/write throughput for other tables.
- The frequency of adjusting the reserved read/write throughput in a calendar day (00:00:00 to 00:00:00 of the second day in UTC time) is unlimited. The adjustment interval must be more than two minutes. Adjusting the reserved read/write throughput of a table is defined as adjusting either the read capacity unit or write capacity unit setting. Such an operation is considered as updating the table.
- A reserved read/write throughput adjustment takes effect within one minute.

The consumed read/write throughput that exceeds the value of the reserved read/write throughput is classified as additional read/write throughput. Costs are calculated based on the unit price of the additional read/write throughput.

Initially, your applications may not have a high throughput. Depending on your business requirements, you can set a low reserved read/write throughput to minimize costs. As your business expands, you can increase the reserved read/write throughput of the table to reflect new business requirements. If you want to quickly import a large volume of data immediately after creating a table, you can set a high reserved write throughput to import the data quickly. After the large volume data import is completed, you can lower the reserved read/write throughput.

Data size restrictions of partition key

Table Store partitions the table data according to the partition key ranges. Rows with the same partition key are placed in the same partition. To prevent large indivisible partitions, we recommend that the total data size for all rows under a single partition key value must not exceed 10 GB.

Table Store load time

Table Store table is ready within one minute after it is created. You must wait for the table to finish loading before performing any data operations.

Best Practice

[See Table operations](#)

Table Store SDKs

[Use Table Store Java SDK for table operations](#)

[Use Table Store Python SDK for table operations](#)

2 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 ($=$, \neq , $>$, \geq , $<$, and \leq) 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
CompositeColumnValueCondition ( CompositeColumnValue
Condition . LogicOperator . AND );
SingleColumnValueCondition single1 = new SingleColumn
ValueCondition ( " Col0 ",
SingleColumnValueCondition . CompareOperator . EQUAL
, ColumnValue . fromLong ( 0 ));
SingleColumnValueCondition single2 = new SingleColumn
ValueCondition ( " 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
CompositeColumnValueCondition ( CompositeColumnValue
Condition . LogicOperator . OR );
SingleColumnValueCondition single3 = new SingleColumn
ValueCondition ( " 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 = PrimaryKey
Builder . createPrimaryKeyBuilder ();
    primaryKeyBuilder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey primaryKey = primaryKeyBuilder . build ();

    // read a row
    SingleRowQueryCriteria criteria = new SingleRowQ
ueryCriteria ( 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 RowUpdateC
hange ( TABLE_NAME , primaryKey );
```

```

        Condition condition = new Condition ( RowExisten
ceExpectat ion . EXPECT_EXI ST );
        ColumnCond ition columnCond ition = new SingleColu
mnValueCon dition (" Col0 ", SingleColu mnValueCon dition .
CompareOpe rator . EQUAL , ColumnValu e . fromLong ( col0Value
));
        condition . setColumnC ondition ( columnCond ition );
        rowUpdateC hange . setCondi ti on ( condition );
        rowUpdateC hange . put ( new Column (" Col0 ", ColumnValu
e . fromLong ( col0Value + 1 )));

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

```

Example

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

```

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

```

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

Billing

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

3 Auto-increment function of the primary key column

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

Features

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

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

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

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

API	IGNORE	EXPECT_EXIST	EXPECT_NOT_EXIST
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.

You cannot change the auto-increment attribute of a table by using UpdateTable.

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

- **GetRow/BatchGetRow**

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

- **Other interfaces**

Not changed

Usage

[Java SDK: Auto-increment of the primary key column](#)

Billing

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

4 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.

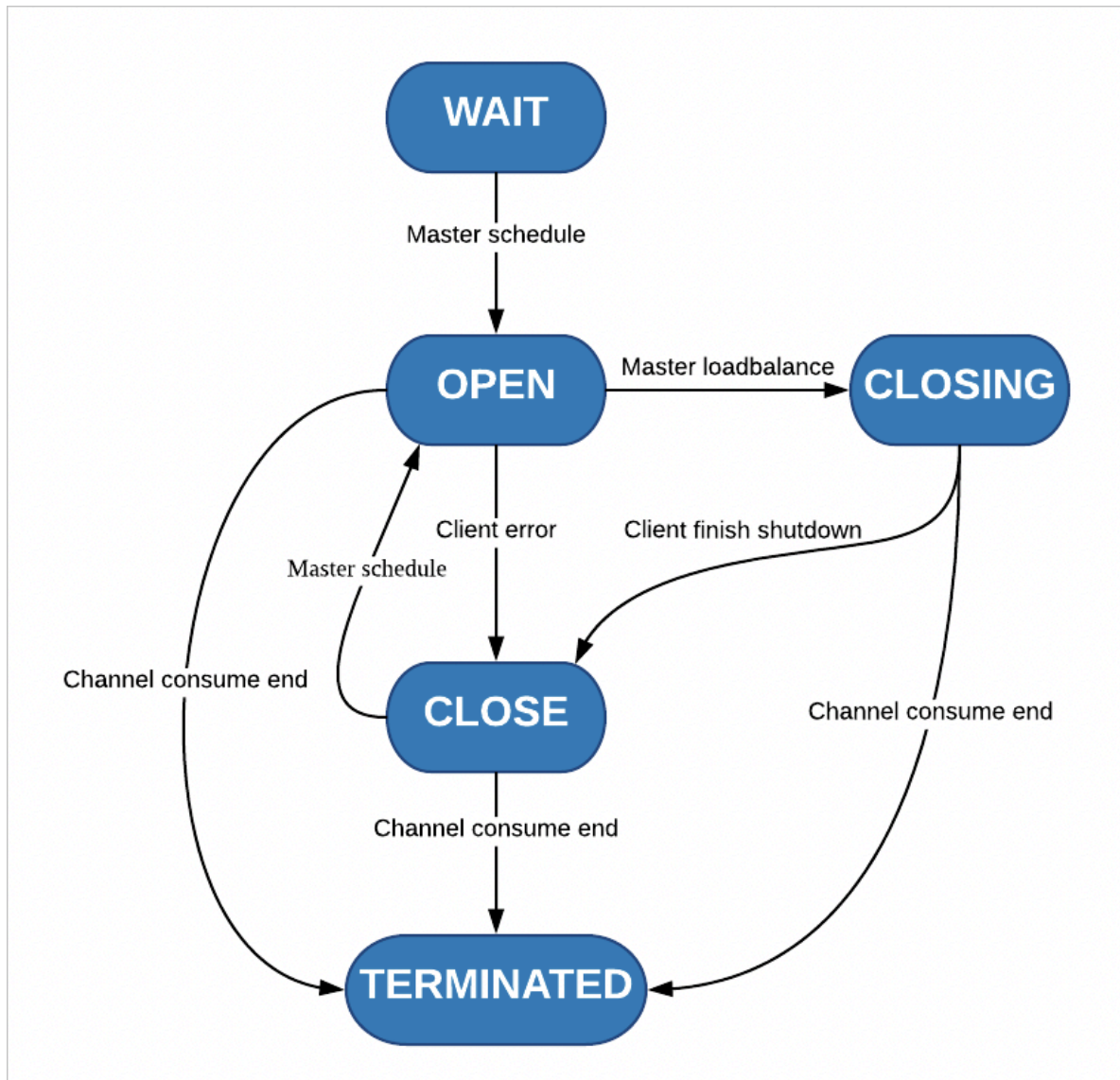
You can set the heartbeatIntervalInSec parameter in TunnelWorkerConfig to set the interval for checking the heartbeat.

- 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.

- Each channel is initially in WAIT status.
- The channel for incremental data changes to the OPEN status only when the channel consumption on the parent partition is terminated.
- Tunnel Service allocates the partition in OPEN status to each Tunnel client.
- 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.

5 Stream

5.1 Overview

Table Store Stream is a data channel that retrieves incremental data from Table Store tables.

You can use the Table Store Stream API to obtain these changes. You can process incremental data streams in real time and replicate changes.

How Stream works

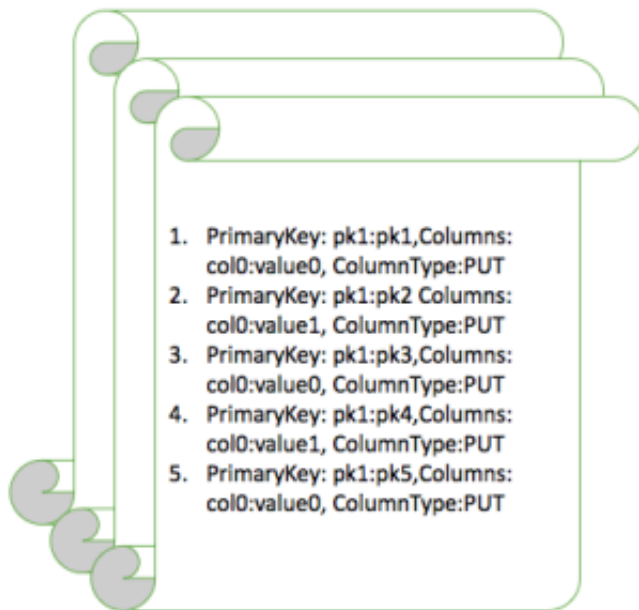
As a distributed NoSQL database, Table Store stores changes in the commit logs of Table Store when executing write operations (including put, delete, and update). Meanwhile, the database also performs regular checkpoints to flush earlier commit log entries.

When Stream is enabled, the log file is retained. The incremental data can be read through the channels provided by Stream during the retention period.

Table Store stores data in shards. Therefore, operations made to the same shard share one commit log. The incremental data is also retrieved at shard level.

When Stream is enabled, the system generates and maintains an offset value (an iterator) to indicate the current read position. You can obtain the iterator of the current shard using the `GetShardIterator` operation. The iterator can be passed in later when you read incremental data stored in this shard. This makes sure that Stream knows which row of log records to read from and return the incremental data. When the incremental data is returned, Stream also returns a new offset for subsequent reads. The whole process can be compared to reading paged data where the iterator is equivalent to the offset of the page.

For example, your database generates some database log files in sequence, as shown in the following figure.



When you enable Stream on row 3 of file A, the iterator points to row 3 of file A. When reading data, you can pass in the iterator to read modifications that occurred after the third operation pk3 in this figure.

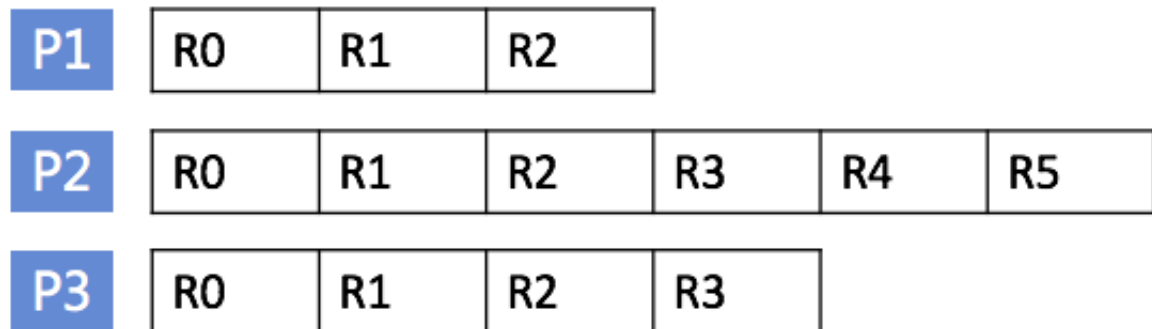
The Stream API also provides an operation to disable this data stream. When you enable it again, Stream generates a new iterator for the current shard, pointing to a new offset that marks the current time. You can use this iterator to read incremental data that occurs after the current time.

Write operations that occur on the same primary key must be read in sequence to guarantee consistency. However, before reading the incremental data, you do not know which primary keys have changes. Therefore, the operation for reading incremental data takes a shard ID and reads from a specific shard. To read the incremental data of the entire table, you can list all shards of the current table. Stream makes sure that write operations made to the same shard are returned in the sequence they were made. In this way, data changes made to a specific shard are read in the same sequence as they were written, and the data consistency for the same primary key can be guaranteed. If you continue to read the Stream data for all the shards, you can make sure that all incremental data in the table is read.

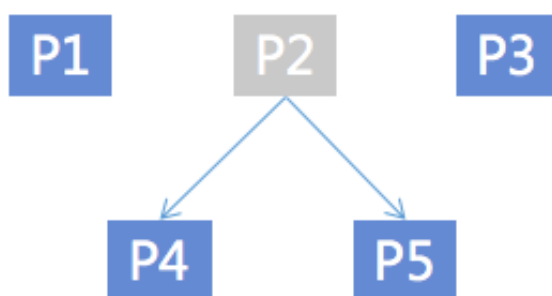
You can either enable Stream when creating a table or enable or disable Stream later using the `UpdateTable` operation. When a `put`, `update`, or `delete` operation occurs, a modification record is written to Stream. The record indicates the primary key values of the row that you modified and the actual modifications.

**Note:**

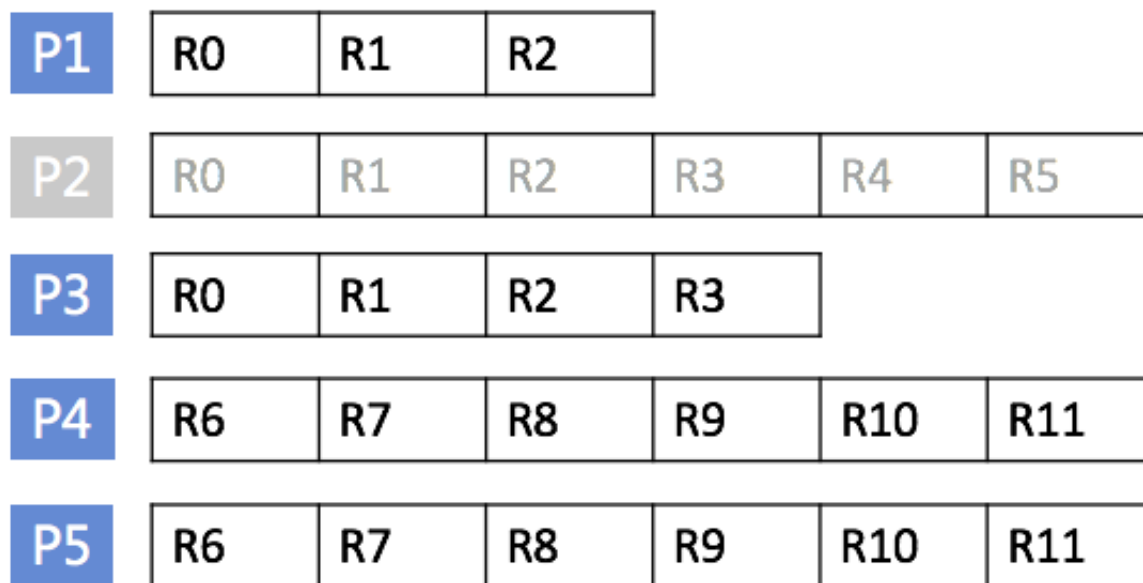
- Each modification record exists in Stream only once.
- For each shard, Stream processes modifications in the sequence they were made. However, modifications made to different shards are not sequenced.

Example

In this figure, the current table has three shards. Each row in this figure represents a shard, and each column represents an update operation on a specific shard. Each shard maintains its own update log. You can use the `DescribeStream` operation to obtain information about the shard, and then read the changes in sequence for this shard. However, the system may split or merge shards in response to varying loads. New shards are created during merge or split operations, and earlier shards no longer generate new incremental data.



In this figure, shard P2 splits into shards P4 and P5. You can read data from shards P4 and P5 in parallel, without affecting one another. However, before you read from shards P4 and P5, make sure that you have read all incremental data on shard P2.



For example, in this figure, when you start reading the R6 log entry of shard P4, make sure that R5 of shard P2 has already been read. After R5 is read, shard P2 does not generate new data.

5.2 Stream API/SDK

API

- Enable and disable Stream

You can specify whether Stream is enabled or disabled when creating a table. Also, you can use the `UpdateTable` operation to enable or disable Stream later. The `CreateTable` and `UpdateTable` operations now include a `StreamSpecification` parameter that allows you to set Stream parameters:

- `enable_stream`: Whether to enable Stream.
- `expiration_time`: Stream data expiration time. Expired modification log entries are deleted.

- Read modification logs

To read Stream data, follow these steps:

1. Call `ListStream s` to obtain the current table' s Stream information, such as Stream ID. For more information, see [ListStream](#).
2. Call `DescribeStream` to obtain the current Stream' s data shard information, such as the shard list. Each shard log contains shard information such as the parent shard and shardID. For more information, see [DescribeStream](#).
3. After obtaining StreamID and shardID, use `GetShardIterator` to obtain the current shard' s read iterator value. This value marks the starting position for reading the shard log. For more information, see [GetShardIterator](#).
4. Call `GetStreamRecord` to read the specific modification logs. Each call returns a new iterator for the next read to use. For more information, see [GetStreamRecord](#).



Notice:

- Operations made to the same primary key have to be sequenced. Stream makes sure that operations made to the same shard are sequenced. However, shards may be split or merged, so before you read the data of a shard, make sure that data of the shard' s parent shard and parent_sibling has been read.
- When an empty `NextShardIterator` is returned, it indicates that incremental data in the current shard has been fully read. This situation occurs typically when the shard is inactive after a split or merge operation. When a shard has been fully read, you can call `DescribeStream` again to retrieve information about the new shard.

SDK

Table Store Java SDK supports the Stream interface. For more information, see [Java SDK](#).

5.3 Stream Client

You can use Table Store Stream APIs and Table Store SDKs to read Stream records. When you obtain incremental data in real time mode, note that information in shards is not static. Shards may be split or merged. When shards are changed, you must

process the dependencies between them to make sure that data in a single primary key is read in sequence. In addition, if your data is generated concurrently from multiple clients, multiple consumers must concurrently read the incremental records in each shard to improve the efficiency of exporting incremental data.

Stream Client is used to resolve common problems during Stream data processing, for example, load balancing, fault recovery, checkpoint, and shard information synchronization to guarantee the information consumption sequence. After using Stream Client, you only need to focus on the processing logic of each record.

This topic describes the principles of Stream Client, and how to use Stream Client to efficiently build a data tunnel that is applicable to your own services.

How Stream Client works

To easily implement job scheduling and record the read progress of each current shard, Stream Client uses a table of Table Store to record the information. You can customize the table name, but you must make sure that this table name is not used by other services.

Stream Client defines a lease for each shard, and the owner of each lease is called the worker. A lease is used to record the incremental data consumers (that is, workers) of the shard and read the progress. When a new consumer is started, the worker is initialized, checking the shard and lease information and creating a lease for a shard if the shard does not have one. When a new shard is generated from shard splitting or combination, Stream Client inserts a lease record into the table. The new record is grabbed and continuously processed by a worker of a Stream Client. If a new worker joins, load balancing is implemented to dispatch the record to the new worker.

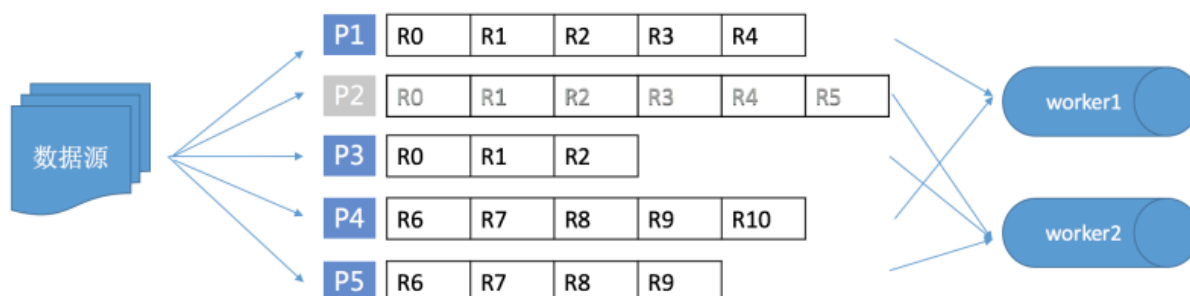
The following table describes the schema of the lease record.

Parameter	Description
Primary key StreamId	ID of the currently processed Stream.
Primary key StatusType	Key of the current lease.
Primary StatusValue	ID of the shard corresponding to the current lease.
Attribute Checkpoint	Location where Stream data is consumed in the current shard (for user fault recovery).

Parameter	Description
Attribute LeaseCounter	Optimistic lock. The owner of each lease continues to update the counter value. Lease renewal indicates that the current lease is continuously occupied.
Attribute LeaseOwner	Name of the worker that owns the current lease.
Attribute LeaseStealer	Worker to which the lease is to be moved during load balancing.
Attribute ParentShardIds	Parent shard of the current shard. When the worker is consuming the current shard, make sure that the Stream of the parent shard has been consumed.

Example

The following figure shows a typical distributed architecture of using Stream Client to consume incremental data.



In this figure, worker1 and worker2 are two consumers based on Stream Client, for example, programs started on the ECS. The data source constantly reads/writes a table in Table Store. In the initial stage, the table contains shards P1, P2, and P3. With the increase of the traffic and data volume, P2 is split into P4 and P5. In the initial stage, worker1 consumes data of P1, and worker2 consumes data of P2 and P3. After P2 is split, P4 will be allocated to worker1, and P5 will be allocated to worker2. However, Stream Client makes sure that data of P4 and P5 is consumed after consumption of record R5 of P2 is complete. If a new consumer worker3 is deployed at this time, a shard on worker2 may be dispatched to worker3, resulting in load balancing.

In the preceding scenario, Stream Client generates the following lease information in the table:

	StreamId	StatusType	StatusValue	Checkpoint	Counter	owner	Stealer	ParentShardIds
P1	Table_123456	LeaseKey	ShardId1	checkpoint1	101	worker1		
P2	Table_123456	LeaseKey	ShardId2	checkpoint2	95	worker2		
P3	Table_123456	LeaseKey	ShardId3	checkpoint3	102	worker2		
P4	Table_123456	LeaseKey	ShardId4	checkpoint4	55	worker1		p2
P5	Table_123456	LeaseKey	ShardId5	checkpoint5	55	worker2		p2

The worker in Stream Client is the carrier of the consumed Stream data. Each shard is allocated to a worker (lease owner). The owner constantly renews the current shard lease through heartbeats, that is, by updating LeaseCounter. Generally, each Stream consumer has a worker. After the worker is initialized, it obtains information about the shard to be processed. At the same time, the worker maintains its own thread pool, and concurrently and cyclically pulls incremental data of each shard it owns. The worker initialization process is as follows:

1. Reads the Table Store configuration and initializes the client that accesses Table Store through an intranet.
2. Obtains the Stream information of the corresponding table and initializes the lease management class. The lease management class synchronizes the lease information and creates a new lease record for the new shard.
3. Initializes the shard synchronization class, which maintain the heartbeats of the current owned shards.
4. Cyclically obtains the incremental data of the shard currently owned by the worker.

Download Stream Client

- Download and install the [JAR package](#)
- Maven:

```
< dependency >
  < groupId > com . aliyun . openservic es </ groupId >
  < artifactId > tablestore - streamclie nt </ artifactId >
  < version > 1 . 0 . 0 </ version >
```

```
</ dependency >
```

**Note:**

The code of Stream Client is open-sourced. You can [download the source code](#) to learn about the principle. You are also welcomed to share good Stream-based sample code with us.

Use Stream Client APIs

Stream Client provides the `IRecordProcessor` API, facilitating you to use Stream Client to consume the Stream data and hide the shard read logic and dispatch logic. The worker of Stream Client calls the `processRecords` function after pulling the Stream data to trigger your data processing logic.

```
public interface IRecordProcessor {
    void initialize ( InitializationInput initializationInput );

    void processRecords ( ProcessRecordsInput processRecordsInput );

    void shutdown ( ShutdownInput shutdownInput );
}
```

The parameters are described as follows:

Parameter	Description
<code>void initialize(InitializationInput initializationInput);</code>	Used to initialize a read task. It indicates that Stream Client is about to read data of a shard.
<code>void processRecords(ProcessRecordsInput processRecordsInput);</code>	Indicates how the user wants to process this batch of records after the data is read. The <code>getCheckpoint</code> function in <code>ProcessRecordsInput</code> can be used to obtain <code>IRecordProcessorCheckpoint</code> . The framework provides this API to implement the checkpoint. You can determine how often the checkpoint is implemented.
<code>void shutdown(ShutdownInput shutdownInput);</code>	Used to end the read task of a shard.

**Note:**

- The read tasks are implemented in different machines, the process may encounter various types of errors, for example, restart due to an environment factor. Therefore, you must periodically record the completed data (checkpoint). When a task is restarted, it is continued from the last checkpoint. In other words, Stream Client does not guarantee that a record is sent through `ProcessRecordsInput` only once. It only guarantees that the record is sent at least once, and that the record sequence does not change. If some data is repeatedly sent, you must pay attention to the service processing logic.
- If you want to reduce the repeat data processing times in case of an error, you can increase the frequency of the checkpoint operation. However, too frequent checkpoints reduce the system throughput. Therefore, determine the checkpoint frequency based on your service features.
- If you find that the incremental data fails to be consumed in time, you can increase resources for the consumer, such as using more nodes to read the Stream record.

The following provides a simple example to describe how to use Stream Client to obtain the incremental data in real time and output the incremental data on the console.

```
public class StreamSample {
    class RecordProcessor implements IRecordProcessor {
        private long creationTime = System.currentTimeMillis();
        private String workerIdentifier;

        public RecordProcessor (String workerIdentifier) {
            this.workerIdentifier = workerIdentifier;
        }

        public void initialize (InitializationInput
        initializationInput) {
            // Trace some info before start the query
            like stream info etc.
        }

        public void processRecords (ProcessRecordsInput
        processRecordsInput) {
            List<StreamRecord> records = processRecordsInput
            .getRecords();

            if (records.size() == 0) {
                // No more records we can wait for the
                next query
                System.out.println("no more records");
            }
        }
    }
}
```

```

    }
    for ( int i = 0 ; i < records . size () ; i ++ ) {
        System . out . println ( " records : " + records . get
( i ) );
    }

    // Since we don ' t persist the stream record
    we can skip blow step
    System . out . println ( processRec ordsInput .
getCheckpo inter (). getLargest PermittedC heckpointV alue () );
    try {
        processRec ordsInput . getCheckpo inter ().
checkpoint ();
    } catch ( ShutdownEx ception e ) {
        e . printStack Trace ();
    } catch ( StreamClie ntExceptio n e ) {
        e . printStack Trace ();
    } catch ( Dependency Exception e ) {
        e . printStack Trace ();
    }
}

public void shutdown ( ShutdownIn put shutdownIn put )
{
    // finish the query task and trace the
    shutdown reason
    System . out . println ( shutdownIn put . getShutdow
nReason () );
}

class RecordProc essorFacto ry implements IRecordPro
cessorFact ory {

    private final String workerIden tifier ;

    public RecordProc essorFacto ry ( String workerIden
tifier ) {
        this . workerIden tifier = workerIden tifier ;
    }

    public IRecordPro cessor createProc essor () {
        return new StreamSamp le . RecordProc essor (
workerIden tifier );
    }
}

public Worker getNewWork er ( String workerIden tifier ) {
    // Please replace with your table info
    final String endPoint = "";
    final String accessId = "";
    final String accessKey = "";
    final String instanceNa me = "";

    StreamConf ig streamConf ig = new StreamConf ig ();
    streamConf ig . setOTSClie nt ( new SyncClient ( endPoint
, accessId , accessKey ,
instanceNa me ));
    streamConf ig . setDataTab leName ( " teststream " );
    streamConf ig . setStatusT ableName ( " statusTabl e " );

    Worker worker = new Worker ( workerIden tifier , new
ClientConf ig (), streamConf ig ,

```



```
        new StreamSample.RecordProcessorFactory (
workerIdentifier), Executors.newCachedThreadPool (), null );
    return worker ;
}

public static void main ( String [] args ) throws
InterruptedException {
    StreamSample test = new StreamSample ();
    Worker worker1 = test . getNewWorker ( " worker1 " );
    Thread thread1 = new Thread ( worker1 );
    thread1 . start ();
}
}
```

6 HBase

6.1 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 . openservice </ 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.

6.2 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](#) functionality, 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.

6.3 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(TableNames tableName)
- getCompactionStateForRegion(byte[] regionName)
- getOnlineRegions(ServerName sn)
- majorCompactRegion(byte[] regionName)
- majorCompactRegion(byte[] regionName, byte[] columnFamily)
- mergeRegions(byte[] encodedNameOfRegionA, byte[] encodedNameOfRegionB, boolean forcible)
- move(byte[] encodedRegionName, byte[] destServerName)
- offline(byte[] regionName)
- splitRegion(byte[] regionName)
- splitRegion(byte[] regionName, byte[] splitPoint)
- stopRegionServer(String hostnamePort)
- unassign(byte[] regionName, boolean force)

Snapshots

Table Store does not support Snapshots related APIs.

Replication

Table Store does not support Replication related APIs.

Coprocessors

Table Store does not support Coprocessors related APIs.

Distributed procedures

Table Store does not support Distributed procedures related APIs.

Table Management

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

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

Restrictions

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

6.4 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 . openservice </ 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 . Tablestore
Connection </ value >
</ property >

```

- **Configuration items of Table Store**

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

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

```

< property >
  < name > tablestore . client . endpoint </ name >
  < value ></ value >
</ property >
< property >
  < name > tablestore . client . 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.

6.5 Migrate HBase of an earlier version

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

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

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

Connection APIs

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

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

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

The latest code is as follows:

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

TableName series

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

The latest code is as follows:

```
String tableName = " MyTable ";
// or byte [] tableName = Bytes . toBytes ( " MyTable " );
TableName tableName0 bj = 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 ();
```

6.6 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 . openservic es </ groupId >
    < artifactId > tablestore - hbase - client </ artifactId >
  >
  < version > 1 . 2 . 0 </ version >
```

```

    </ dependency >
  </ dependenci es >

```

For more information about advanced configurations, see [Migrate from HBase to Table Store](#).

- **Configure the file**

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

```

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

```

For more information about advanced configurations, see [Migrate from HBase to Table Store](#).

- **Connect Table Store**

Create a `TableStoreConnection` object to connect Table Store.

```

Configurat ion config = HBaseConfi guration . create ();

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

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

```

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

- **Create a table**

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

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

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

// Add a column . Table Store supports only single
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 = Connection Factory .
createConnection ( config );
        Admin admin = connection . getAdmin ();

        HTableDescriptor descriptor = new HTableDesc
ription ( TableName . valueOf ( TABLE_NAME ));
        descriptor . addFamily ( new HColumnDes cription (
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 );  
    }  
}
```


7 SearchIndex

7.1 Overview

Searchindex is an advanced index structure. Searchindex supports multiple advanced queries, including the ad-hoc query, fuzzy query, full-text retrieval, sorting, range query, nested query, and spatial query.

Synchronize indexes

After you enable the Searchindex feature for a table, incremental data written to the table is synchronized to the storage module of Table Store, and then Table Store indicates the successful write operation in the response. Meanwhile, an asynchronous thread reads incremental data from the storage module and creates a Searchindex structure. This asynchronous process has a latency from several milliseconds up to a few seconds, but does not affect the write performance of Table Store.

Restrictions

Compared with the KeyValue storage module of Table Store, Searchindex does not support the following features.

Feature	Description
TTL	You cannot create a Searchindex structure in a table where you have specified the Time To Live (TTL) parameter.
Max Versions	You cannot create a Searchindex structure in a table where you have specified the Max Versions parameter.
Custom versions	You can customize the timestamp whenever writing data to an attribute that allows only one version. If you first write a large version number and then a small version number, the index of the large version number may be overwritten with the index of the small version number.

Features

Searchindex provides the following features:

- Ad-hoc query

- Fuzzy query
- Range query
- Full-text retrieval
- Spatial query
- Prefix query
- Nested query

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

Usage

Table Store provides Searchindex API operations and SDKs in multiple languages to allow you easily use Searchindex.

Scenarios

Searchindex allows Table Store to be applicable to multiple scenarios, such as metadata, spatio-temporal data, time series data, and full-text retrieval.

Billing

The metering and billing of index tables is separated from primary tables.

7.2 Features

Table Store provides the Searchindex structure in addition to the primary key prefix query to help you easily search data.

Searchindex supports the following indexes:

- **Inverted index:** a basic data structure of multiple queries in a search engine. This data structure improves query performance. If you create inverted indexes for some attribute columns, Table Store can use the ad-hoc query that flexibly combines multiple fields. Therefore, you can have more options besides the fields such as gender, age, and enumeration.
- **Multi-dimensional spatial index:** a data structure used in a geographic information system (GIS) or other multi-dimensional spatial queries. This structure is usually used in spatio-temporal data scenarios to optimize the spatial query performance. Table Store can use the multi-dimensional spatial index to query a range of geographical points such as nearby persons, points in a rectangle, and points in a polygon. The system works as a powerful comprehensive data processing platform for big data filtering, Internet of Vehicles (IoV), and mobile apps.

- **Columnstore forward index:** a data structure suitable for database features such as sorting, statistics, and aggregation.

Searchindex supports the following features:

Non-primary key query

Searchindex supports the query by non-primary keys. You can call the `CreateSearchIndex` operation to create the Searchindex structure for the required column and search data by the value of the column.

Ad-hoc query

Neither NoSQL databases nor relational databases can perform an excellent ad-hoc query. For example, for a table that has four columns, you may create seven secondary indexes in relational databases if you do not know the correct query method or have to search multiple fields. If the table is added with one more column, the query may require more than 10 indexes. However, Searchindex only requires the existing four columns and supports relational operators such as `&&` (AND), `(OR)`, and `!` (NOT).

GIS

A GIS is a powerful feature used in geographical applications such as IoV and IoT. For example, you can use the GIS to search nearby vehicles and points in a polygon such as an electric fence. Table Store provides the Searchindex feature, so you can use the GIS to search nearby vehicles within a radius of N km, points in a rectangle, or points in a polygon. In this way, you can conveniently develop big data applications based on the GIS.

Sorting

Table Store supports multiple types of sorting, such as forward sorting, reverse sorting, single-field sorting, and multi-field sorting, so you can easily sort global data.

Full-text retrieval

Table Store can tokenize data based on inverted indexes and support full-text retrieval. You can recall data, but cannot perform any correlation analysis. The system tokenizes data in two ways: `single_word` and `max_word`.

Fuzzy query

Table Store enables the fuzzy query. You can create an inverted index for an attribute column to start the fuzzy query in this field.

Prefix query

You can create an inverted index for an attribute column to start the prefix query in this field.

Nested query

The nested query allows you to easily build models from complex data.

Such online data as pictures and labels have some complex multi-layered structures . For example, a database stores a large number of pictures, and each picture has multiple elements, such as houses, cars, or people. Each element in a picture has a unique score. The score is evaluated according to the size and position of an element in a picture. Therefore, each picture has multiple labels. Each label has a name and a weighted score. You can use the nested query based on the conditions or field names of the labels.

Cardinality

If an attribute contains a large amount of data, the query result based on this attribute may have many repeats. However, cardinality allows you to specify the highest frequency of occurrence of an attribute value to achieve high cardinality.

Total number of rows

Table Store returns the total number of rows in a result set whenever you apply Searchindex. If you do not specify any query condition for Searchindex, 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.

If any issues occur when you use Searchindex, you can request customer service by using any of the following methods:

- [Submit a ticket in the Table Store console.](#)
- Join the DingTalk group of Table Store technical support: 11789671.

8 Global secondary indexes

8.1 Overview

Before you start using global secondary indexes, you need to understand the following basic concepts, restrictions, and precautions.

Concepts

Terminology	Description
Index	You can create an index on some attributes of a table. An index is read-only.
Pre-defined attribute	Table Store uses a schema-free model. You can write the unlimited number of attributes to a row instead of specifying a fixed number of attributes in a schema . You can also specify pre-defined attributes and their data types when you create a table.
Single-field index	You can only create an index on one table attribute.
Compound index	You can create an index on multiple attribute groupings of a table. In addition , you can sort by these attributes.
Projected attribute	You can copy pre-defined attributes to an index. None of these attributes are specified as a primary key.
Auto-complement	The primary keys of a table are copied to an index automatically when you create an index.

Restrictions

- You can create a maximum of 16 indexes on a table.
- An index can have a maximum of four indexed attributes. These attributes consist of base table primary keys and pre-defined attributes.

- You can specify Integer, String, or Binary for an indexed attribute as its data type. The attribute restrictions are the same as those for a primary key.
- When you specify Integer or Binary for an attribute, the size restrictions of these data types are the same as those for a base table primary key.
- When you specify String or Binary for an attribute, the attribute restrictions are the same as those for a base table.
- You cannot create an index on a table that is configured with the TTL property. If you want to create such indexes, contact Table Store technical support by DingTalk .
- You cannot create an index on a table that is configured with the Max Versions property.
- You cannot use Stream on an index.
- You cannot create an index on a table with an auto increment primary key. If you have such indexes, open a ticket to Table Store technical support.

Precautions

- Table Store will perform auto-complement operations on each index. When you scan an index, you must specify a primary key value range. In general, the range is from negative to positive infinity. For example, a table includes the primary keys `PK0` and `PK1` and the pre-defined attribute `Defined0` .

When you create an index on the `Defined0` attribute, Table Store will generate an index that have `Defined0` , `PK0` , and primary keys `PK1` . When you create an index on the `Defined0` and `PK1` attributes, Table Store will build an index that have `Defined0` , and primary keys `PK1` , `PK0` . When you create an index on the `PK` attribute, Table Store will build an index with the primary keys `PK1` and `PK0` . When you create an index, you can only specify attributes to be indexed. Table Store will perform auto-complement operations on the index later. For example, a table with primary keys `PK0` and `PK1`, and the pre-defined attribute `Defined0`.

- When you create an index on the `Defined0` attribute, Table Store will build the index that have the primary keys `Defined0`, `PK0`, and `PK1`.
- When you create an index on the `PK1`, Table Store will build the index that has the primary keys `PK0` and `PK1`.

- You can specify a table pre-defined attributes as index attributes. When you specify a pre-defined attribute as an index attribute, you can query this index to read the attribute value instead of querying the base table. This configuration will increase storage costs. If you did not specify a pre-defined attribute as an index attribute, you must query the base table. You can select a query mode based on your requirements and costs.
- We do not recommend that you specify an attribute related to the time or date as the first attribute of primary keys of an index because the configuration may lead to slow index updates. We recommend that you enter an attribute related to the time or date as the input for a hash function and create an index on the attribute that has been processed by the hash function. If you have such requirements, contact Table Store technical support by DingTalk.
- We do not recommend you define an attribute with low cardinality, even an attribute with enumerable values, as the first attribute of primary keys of an index. For example, the configured value `gender` restricts the index horizontal scalability and leads to poor data writing performance.

8.2 Introduction

A global secondary index in Table Store 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. In later versions , 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.

The Table Store global secondary index feature is now available in China (Zhangjiakou) region. You can contact Table Store technical support by DingTalk for a trial or enter the ID 111789671 to join the DingTalk group for further information.

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
RangeRowQueryCriteria ( TABLE_NAME );

    // You can specify primary keys .
    PrimaryKeyBuilder startPrimaryKeyBuilder =
PrimaryKeyBuilder.createPrimaryKeyBuilder ();
    startPrimaryKeyBuilder.addPrimary KeyColumn (
PRIMARY_KEY_NAME_1, PrimaryKey Value . fromLong ( cellNumber
));
    startPrimaryKeyBuilder.addPrimary KeyColumn (
PRIMARY_KEY_NAME_2, PrimaryKey Value . fromLong ( 0 ));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey (
startPrimaryKeyBuilder.build ());

    // You can specify primary keys .
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKey
Builder.createPrimaryKeyBuilder ();
    endPrimaryKeyBuilder.addPrimary KeyColumn ( PRIMARY_KEY
NAME_1, PrimaryKey Value . fromLong ( cellNumber ));
    endPrimaryKeyBuilder.addPrimary KeyColumn ( PRIMARY_KEY
NAME_2, PrimaryKey Value . INF_MAX );
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey (
endPrimaryKeyBuilder.build ());

    rangeRowQueryCriteria.setMaxVersions ( 1 );

    String strNum = String . format ("% d ", cellNumber );
    System . out . println ( " A cell number " + strNum + "
makes the following calls : " );
}
```

```

while ( true ) {
    GetRangeResponse response = client .
    getRange ( new GetRangeRequest ( rangeRowQueryCriteria ) );
    for ( Row row : response . getRows () ) {
        System . out . println ( row );
    }

    // If the value of nextStartPrimaryKey is not
    null , you can continue to read data from the
    base table .
    if ( response . getNextStartPrimaryKey () !=
    null ) {
        rangeRowQueryCriteria . setInclusiveStartPri
        maryKey ( response . 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 a = new RangeRowQueryCriteria ( INDEX0_NAME );

    // You can specify primary keys .
    PrimaryKeyBuilder startPrimaryKeyBuilder =
    PrimaryKeyBuilder.createPrimaryKeyBuilder ();
    startPrimaryKeyBuilder.addPrimary KeyColumn (
    DEFINED_COLUMN_NAME_1 , PrimaryKey Value . fromLong ( cellNumber
    ));
    startPrimaryKeyBuilder.addPrimary KeyColumn (
    PRIMARY_KEY_NAME_1 , PrimaryKey Value . INF_MIN );
    startPrimaryKeyBuilder.addPrimary KeyColumn (
    PRIMARY_KEY_NAME_2 , PrimaryKey Value . INF_MAX );
    rangeRowQueryCriteria a . setIncludeStartPrimaryKey (
    startPrimaryKeyBuilder . build ());

    // You can specify primary keys .
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKey
    Builder.createPrimaryKeyBuilder ();
    endPrimaryKeyBuilder.addPrimary KeyColumn ( DEFINED_CO
    L_NAME_1 , PrimaryKey Value . fromLong ( cellNumber ));
    endPrimaryKeyBuilder.addPrimary KeyColumn ( PRIMARY_KEY
    NAME_1 , PrimaryKey Value . INF_MAX );
    endPrimaryKeyBuilder.addPrimary KeyColumn ( PRIMARY_KEY
    NAME_2 , PrimaryKey Value . INF_MAX );
    rangeRowQueryCriteria a . setExcludeEndPrimaryKey (
    endPrimaryKeyBuilder . build ());

    rangeRowQueryCriteria a . setMaxVersions ( 1 );

    String strNum = String . format ("% d ", cellNumber );
    System . out . println (" A cell number " + strNum + "
    was called by the following numbers ");
    while ( true ) {
        GetRangeResponse response = client .
        getRange ( new GetRangeRequest ( rangeRowQueryCriteria a ));
        for ( Row row : response . getRows () ) {
            System . out . println ( row );
        }

        // If the value of nextStartPrimaryKey is not
        null , you can continue to read data from the
        base table .
        if ( response . getNextStartPrimaryKey () !
        = null ) {
            rangeRowQueryCriteria a . setIncludeStartPri
            maryKey ( response . 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 baseStation
nNumber ,
                                         long startTime ) {
    RangeQueryCriteria rangeQueryCriteria = new
RangeQueryCriteria ( INDEX1_NAME );

    // You can specify primary keys .
    PrimaryKeyBuilder startPrimaryKeyBuilder =
PrimaryKeyBuilder.createPrimaryKeyBuilder ();
    startPrimaryKeyBuilder.addPrimary KeyColumn (
DEFINED_COLUMN_NAME_3 , PrimaryKey Value . fromLong ( baseStation
nNumber ));
    startPrimaryKeyBuilder.addPrimary KeyColumn (
PRIMARY_KEY_NAME_2 , PrimaryKey Value . fromLong ( startTime
));
    startPrimaryKeyBuilder.addPrimary KeyColumn (
PRIMARY_KEY_NAME_1 , PrimaryKey Value . INF_MIN );
    rangeQueryCriteria.setInclusiveStartPrimaryKey (
startPrimaryKeyBuilder.build ());

    // You can specify primary keys .
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKey
Builder.createPrimaryKeyBuilder ();
    endPrimaryKeyBuilder.addPrimary KeyColumn ( DEFINED_CO
L_NAME_3 , PrimaryKey Value . fromLong ( baseStation nNumber ));

```

```

        endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KEY_NAME_2 , PrimaryKey Value . INF_MAX );
        endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KEY_NAME_1 , PrimaryKey Value . 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 . setInclusiveStartPri
maryKey ( getRangeResponse . getNextStartPrimaryKey ());
            } else {
                break ;
            }
        }
    }
}

```

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

In this query, you specify both 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 getRowFrom IndexAndMainTable (
SyncClient client ,
                                long baseStation
nNumber ,
                                long startTime ,
                                long endTime ,
                                String colName )
{
    RangeRowQueryCriteria rangeRowQueryCriteria = new
RangeRowQueryCriteria ( INDEX1_NAME );

    // You can specify primary keys .
    PrimaryKeyBuilder startPrimaryKeyBuilder =
PrimaryKeyBuilder . createPrimaryKeyBuilder ();

```

```

        startPrimaryKeyBuilder.addPrimaryKeyColumn(
            DEFINED_COLUMN_NAME_3, PrimaryKeyValue.fromLong(baseStationNumber));
        startPrimaryKeyBuilder.addPrimaryKeyColumn(
            PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong(startTime));
        startPrimaryKeyBuilder.addPrimaryKeyColumn(
            PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MIN);
        rangeQueryCriteria.setInclusiveStartPrimaryKey(
            startPrimaryKeyBuilder.build());

        // You can specify primary keys.
        PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
        endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COLUMN_NAME_3, PrimaryKeyValue.fromLong(baseStationNumber));
        endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong(endTime));
        endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX);
        rangeQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());

        rangeQueryCriteria.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(rangeQueryCriteria));
            for (Row row : fig.getRows()) {
                PrimaryKey currentIndexPrimaryKey = row.getPrimaryKey();
                PrimaryKeyColumn mainCalledNumber = currentIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_NAME_1);
                PrimaryKeyColumn callStartTime = currentIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
                PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
                mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, mainCalledNumber.getValue());
                mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_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 SingleRowQueryCriteria(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);
            }
        }

```

```

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

        System . out . println ( mainTableR ow );
    }

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

```

To improve query performance, you can create a compound index on

`BaseStatio nNumber` and `StartTime` . You can specify `Duration` as an attribute of this index.

The following index is created.

`IndexOnBas eStation2` :

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

You can query the `IndexOnBas eStation2` index:

```

private static void getRangeFr omIndexTab le ( SyncClient
client ,
                                long baseStatio
nNumber ,
                                long startTime ,
                                long endTime ,
                                String colName ) {
    RangeRowQu eryCriteri a rangeRowQu eryCriteri a = new
RangeRowQu eryCriteri a ( INDEX2_NAM E );
    // You can specify primary keys .
}

```



```

    PrimaryKey Builder startPrimaryKeyBuilder =
    PrimaryKey Builder . createPrimaryKeyBuilder ();
    startPrimaryKeyBuilder . addPrimaryKeyColumn (
    DEFINED_COLUMN_NAME_3 , PrimaryKey Value . fromLong ( baseStationNumber ));
    startPrimaryKeyBuilder . addPrimaryKeyColumn (
    PRIMARY_KEY_NAME_2 , PrimaryKey Value . fromLong ( startTime
    ));
    startPrimaryKeyBuilder . addPrimaryKeyColumn (
    PRIMARY_KEY_NAME_1 , PrimaryKey Value . INF_MIN );
    rangeQueryCriteria . setInclusiveStartPrimaryKey (
    startPrimaryKeyBuilder . build ());

    // You can specify primary keys .
    PrimaryKey Builder endPrimaryKeyBuilder = PrimaryKey
    Builder . createPrimaryKeyBuilder ();
    endPrimaryKeyBuilder . addPrimaryKeyColumn ( DEFINED_COLUMN_NAME_3 , PrimaryKey Value . fromLong ( baseStationNumber ));
    endPrimaryKeyBuilder . addPrimaryKeyColumn ( PRIMARY_KEY_NAME_2 , PrimaryKey Value . fromLong ( endTime ));
    endPrimaryKeyBuilder . addPrimaryKeyColumn ( PRIMARY_KEY_NAME_1 , PrimaryKey Value . INF_MAX );
    rangeQueryCriteria . setExclusiveEndPrimaryKey (
    endPrimaryKeyBuilder . build ());

    // You can specify the attribute name to read .
    rangeQueryCriteria . addColumnsToGet ( columnName );

    rangeQueryCriteria . 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 ( rangeQueryCriteria ));
        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 ) {
            rangeQueryCriteria . 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 `BaseStationNumber` `003` with a `StartTime` value range from `1532574861` to `1532584054`.

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

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 . addPrimary KeyColumn ( new PrimaryKey Schema (
PRIMARY_KEY_NAME_1 , PrimaryKey Type . STRING )); // You can
specify a primary key for a base table .
    tableMeta . addPrimary KeyColumn ( new PrimaryKey Schema (
PRIMARY_KEY_NAME_2 , PrimaryKey Type . INTEGER )); // 为主表设置
PK 列
    tableMeta . addDefined Column ( new DefinedColumnSchema (
DEFINED_COLUMN_NAME_1 , DefinedColumn Type . STRING )); // You
can specify a pre - defined attribute for the base
table .
    tableMeta . addDefined Column ( new DefinedColumnSchema (
DEFINED_COLUMN_NAME_2 , DefinedColumn Type . INTEGER )); // You
can specify a pre - defined attribute for the base
table .
    tableMeta . addDefined Column ( new DefinedColumnSchema (
DEFINED_COLUMN_NAME_3 , DefinedColumn Type . INTEGER )); // You
can specify a pre - defined attribute for the base
table .
}
```

```

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

        TableOptions tableOptions = new TableOptions (
timeToLive , maxVersion s );

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

        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 ); //
You can create an index .
    indexMeta . addPrimary KeyColumn ( DEFINED_CO L_NAME_2 ); //
You can specify DEFINED_CO L_NAME_2 as the first
attribute of an index primary key .
    indexMeta . addPrimary KeyColumn ( DEFINED_CO L_NAME_1 ); //
You can specify DEFINED_CO L_NAME_1 as the second
attribute of an index primary key .
    CreateIndexRequest request = new CreateIndexRequest (
TABLE_NAME , indexMeta , false ); // You can create an
index on a base table .
    client . createIndex ( request ); // You can create an
index .
}

```



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 ) {
    DeleteIndex request = new DeleteIndex (
        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 ) {
    RangeQueryCriteria rangeQueryCriteria = new
    RangeQueryCriteria ( INDEX_NAME ); // You can specify
    the name of an index .

    // You can specify the start primary key .
    PrimaryKeyBuilder startPrimaryKeyBuilder =
    PrimaryKeyBuilder . createPrimaryKeyBuilder ();
    startPrimaryKeyBuilder . addPrimaryKeyColumn (
    DEFINED_COLUMN_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 .
    rangeQueryCriteria . setInclusiveStartPrimaryKey (
    startPrimaryKeyBuilder . build ());

    // You can specify the end primary key .
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKey
    Builder . createPrimaryKeyBuilder ();
    endPrimaryKeyBuilder . addPrimaryKeyColumn ( DEFINED_CO
    L_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 .
    rangeQueryCriteria . setExclusiveEndPrimaryKey (
    endPrimaryKeyBuilder . build ());

    rangeQueryCriteria . setMaxVersions ( 1 );

    System . out . println ( " The results returned from an
    index are as follows :");
    while ( true ) {
        GetRangeResponse getRangeResponse = client .
        getRange ( new GetRangeRequest ( rangeQueryCriteria ));
        for ( Row row : getRangeResponse . getRows ()) {
```

```

        System . out . println ( row );
    }

    // If the nextStartPrimary key value is not
    null , you can continue to read data from the base
    table .
    if ( getRangeResponse . getNextStartPrimary key () !
    = null ) {
        rangeRowQueryCriteria . setInclusiveStartPri
        maryKey ( getRangeResponse . getNextStartPrimary key ());
    } 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
    RangeRowQueryCriteria ( INDEX_NAME ); // You can specify
    the index name .

    // You can specify the start primary key .
    PrimaryKeyBuilder startPrimaryKeyBuilder =
    PrimaryKeyBuilder . createPrimaryKeyBuilder ();
    startPrimaryKeyBuilder . addPrimary KeyColumn (
    DEFINED_COLUMN_NAME_1 , PrimaryKey Value . INF_MIN ); // You
    can specify the minimum value for an indexed
    attribute of an index .
    startPrimaryKeyBuilder . addPrimary KeyColumn (
    PRIMARY_KEY_Y_NAME_1 , PrimaryKey Value . INF_MIN ); // You
    can specify the minimum value for a primary key
    of a base table .
    startPrimaryKeyBuilder . addPrimary KeyColumn (
    PRIMARY_KEY_Y_NAME_2 , PrimaryKey Value . INF_MIN ); // You
    can specify the minimum value for a primary key
    of a base table .
    rangeRowQueryCriteria . setInclusiveStartPrimary key (
    startPrimaryKeyBuilder . build ());

    // You can specify the end primary key .
    PrimaryKeyBuilder endPrimary KeyBuilder = PrimaryKey
    Builder . createPrimaryKeyBuilder ();
    endPrimary KeyBuilder . addPrimary KeyColumn ( DEFINED_CO
    L_NAME_1 , PrimaryKey Value . INF_MAX ); // You can specify
    the maximum value for an indexed attribute of an
    index .
    endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KEY
    Y_NAME_1 , PrimaryKey Value . INF_MAX ); // You can specify
    the maximum value for a base table primary key .
    endPrimary KeyBuilder . addPrimary KeyColumn ( PRIMARY_KEY
    Y_NAME_2 , PrimaryKey Value . INF_MAX ); // You can specify
    the maximum value for a base table primary key .
    rangeRowQueryCriteria . setExclusiveEndPrimary key (
    endPrimary KeyBuilder . build ());

    rangeRowQueryCriteria . setMaxVersions ( 1 );

    while ( true ) {

```

```

        GetRangeResponse sponse = client .
getRange ( new GetRangeRequest ( rangeRowQueryCriteria ));
        for ( Row row : getRangeResponse . getRows ()) {
            PrimaryKey curIndexPrimaryKey = row . getPrimary
Key ();
            PrimaryKey Column pk1 = curIndexPrimaryKey .
getPrimary KeyColumn ( PRIMARY_KEY_Y_NAME1 );
            PrimaryKey Column pk2 = curIndexPrimaryKey .
getPrimary KeyColumn ( PRIMARY_KEY_Y_NAME2 );
            PrimaryKey Builder mainTablePKBuilder =
PrimaryKey Builder . createPrimaryKeyBuilder ();
            mainTablePKBuilder . addPrimary KeyColumn (
PRIMARY_KEY_Y_NAME1 , pk1 . getValue ());
            mainTablePKBuilder . addPrimary KeyColumn (
PRIMARY_KEY_Y_NAME2 , pk2 . 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
SingleRowQueryCriteria ( TABLE_NAME , mainTablePK );
            criteria . addColumns ToGet ( DEFINED_COLUMN_NAME3
); // You can read the DEFINED_COLUMN_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 . setInclusiveStartPri
maryKey ( 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 Billing rules

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



Note:

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

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

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

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

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

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

Calculate the number of read CUs consumed to write index tables

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

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

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

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

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

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

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



Note:

We recommend that you use primary key auto increment to insert data to a primary table to decrease the number of CUs that are consumed by index tables.

For primary tables that do not use primary key auto increment, one read CU is consumed if a read operation is performed on the indexed columns, even if no data is retrieved. For primary tables that use primary key auto increment, no read operation is performed on the indexed columns when you insert data. Therefore, no read CU is consumed.

Calculate the number of write CUs

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

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

The detailed rules are as follows:

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

Divide the total amount of data in the inserted index row by four. The number of consumed CUs equals the calculated value rounded up to the nearest integer.

- When you use the PUT operation to overwrite a row in the primary table:
 - The PUT operation only updates the indexed primary key columns in the primary table. In this case, no write CUs are consumed.
 - The PUT operation updates the indexed columns in the primary table. The write CUs are consumed based on the following rules:

All indexes updated by the PUT operation consume a certain number of write CUs, except sparse indexes.

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

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

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

Measure index table size

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

Calculate the number of CUs consumed to read an index table

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

Examples

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

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

.

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

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

- Updating Col1 consumes the following CUs:

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

8.7 Appendix

You can create tables and indexes as follows:

```
private static final String TABLE_NAME = " CallRecord Table ";
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_COLUMN_NAME_1 = "
CalledNumber ";
private static final String DEFINED_COLUMN_NAME_2 = "
Duration ";
private static final String DEFINED_COLUMN_NAME_3 = "
BaseStationNumber ";

private static void createTable ( SyncClient client ) {
    TableMeta tableMeta = new TableMeta ( TABLE_NAME );
    tableMeta.addPrimaryKeyColumn ( new PrimaryKeySchema
( PRIMARY_KEY_NAME_1, PrimaryKeyType.INTEGER ));
    tableMeta.addPrimaryKeyColumn ( new PrimaryKeySchema
( PRIMARY_KEY_NAME_2, PrimaryKeyType.INTEGER ));
    tableMeta.addDefinedColumn ( new DefinedColumnSchema
( DEFINED_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_1);
        indexMetas.add(indexMeta0);
        IndexMeta indexMeta1 = new IndexMeta(INDEX1_NAME);
        indexMeta1.addPrimaryKeyColumn(DEFINED_COLUMN_3);
        indexMeta1.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMetas.add(indexMeta1);
        IndexMeta indexMeta2 = new IndexMeta(INDEX2_NAME);
        indexMeta2.addPrimaryKeyColumn(DEFINED_COLUMN_3);
        indexMeta2.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMeta2.addDefinedColumn(DEFINED_COLUMN_2);
        indexMetas.add(indexMeta2);

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

        client.createTable(request);
    }

```