Alibaba Cloud Table Store

Functions

Issue: 20181106

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.	 **Note:** Take the necessary precautions to save exported data containing sensitive information.
	This indicates supplemental instructions, best practices, tips, and other content that is good to know for the user.	 **Note:** You can use **Ctrl** + **A** to select all files.
>	Multi-level menu cascade.	**Settings** > **Network** > **Set network type**
Bold	It is used for buttons, menus, page names, and other UI elements.	Click **OK**.
`Courier font`	It is used for commands.	Run the `cd /d C:/windows` command to enter the Windows system folder.
Italics	It is used for parameters and variables.	`bae log list --instanceid` *`Instance_ID`*
[] or [a|b]	It indicates that it is a optional value, and only one item can be selected.	`ipconfig` *`[-all|-t]`*
{} or {a|b}	It indicates that it is a required value, and only one item can be selected.	`swich` *`{stand | slave}`*

Contents

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 (=, ! =, >, >=, <, and <=) and logical operations (NOT, AND, and OR). The conditional update is applicable to *PutRow*, *UpdateRow*, *DeleteRow*, and *BatchWriteRow*.

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

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

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

Note:

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

Procedure

1. Construct SingleColumnValueCondition.

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

```
 singleColumnValueCondition.setLatestVersionsOnly(true);
```

2. Construct CompositeColumnValueCondition.

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

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

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

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

     // read a row
     SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(
TABLE_NAME, primaryKey);
     criteria.setMaxVersions(1);
     GetRowResponse getRowResponse = client.getRow(new GetRowRequest
(criteria));
     Row row = getRowResponse.getRow();
     long col0Value = row.getLatestColumn("Col0").getValue().asLong
();

     // Col0 = Col0 + 1 by conditional update
     RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);
     Condition condition = new Condition(RowExistenceExpectation.
EXPECT_EXIST);
     ColumnCondition columnCondition = new SingleColumnValueCon
dition("Col0", SingleColumnValueCondition.CompareOperator.EQUAL,
ColumnValue.fromLong(col0Value));
     condition.setColumnCondition(columnCondition);
```

```
        rowUpdateChange.setCondition(condition);
        rowUpdateChange.put(new Column("Col0", ColumnValue.fromLong(
col0Value + 1)));

        try {
            client.updateRow(new UpdateRowRequest(rowUpdateChange));
        } catch (TableStoreException ex) {
            System.out.println(ex.toString());
        }
    }
```

Example

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

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

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

Billing

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

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
DeleteRow: The row exists.	Fail	Fail	Fail
DeleteRow: The row does not exist.	Succeed	Succeed	Fail

Limits

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

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

Interface

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

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

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

  Not changed

Usage

Java SDK: Auto-increment of the primary key column

Billing

The auto-increment function of primary key columns does not affect the existing billing logic.

Returned data of the primary key column does not consume additional read CUs.

4 Stream

4.1 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

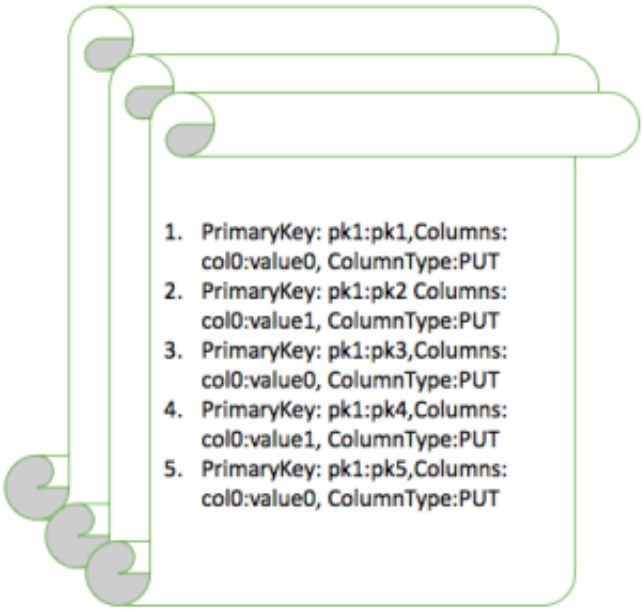
As a distributed NoSQL database, Table Store stores changes in the commit logs of Table Store when executing write operations (including put, delete, and update). Meanwhile, the database also performs regular checkpoints to flush earlier commit log entries.

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.

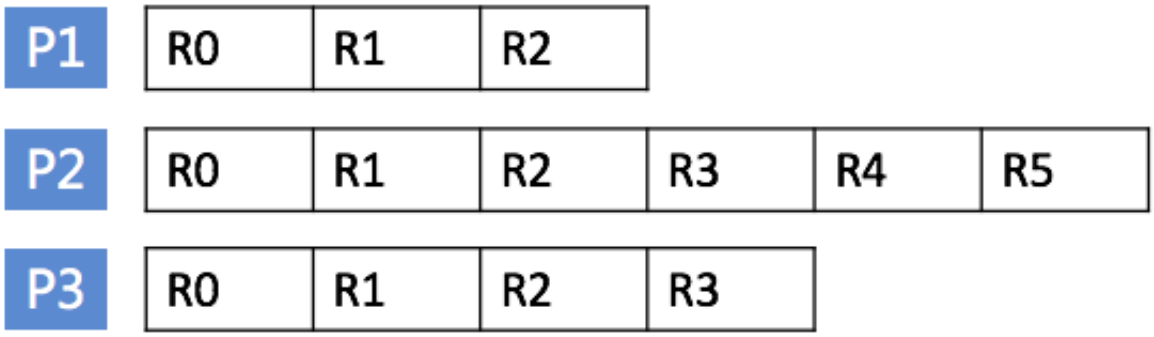
Write operations that occur on the same primary key must be read in sequence to guarantee consistency. However, before reading the incremental data, you do not know which primary keys have changes. Therefore, the operation for reading incremental data takes a shard ID and reads from a specific shard. 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.

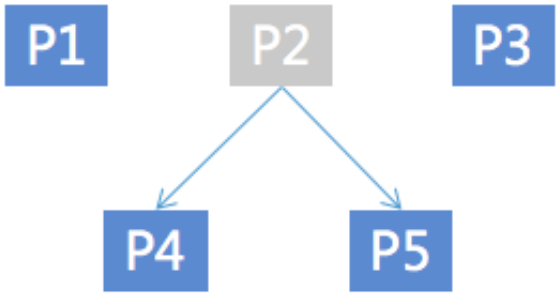
Note:

- 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.

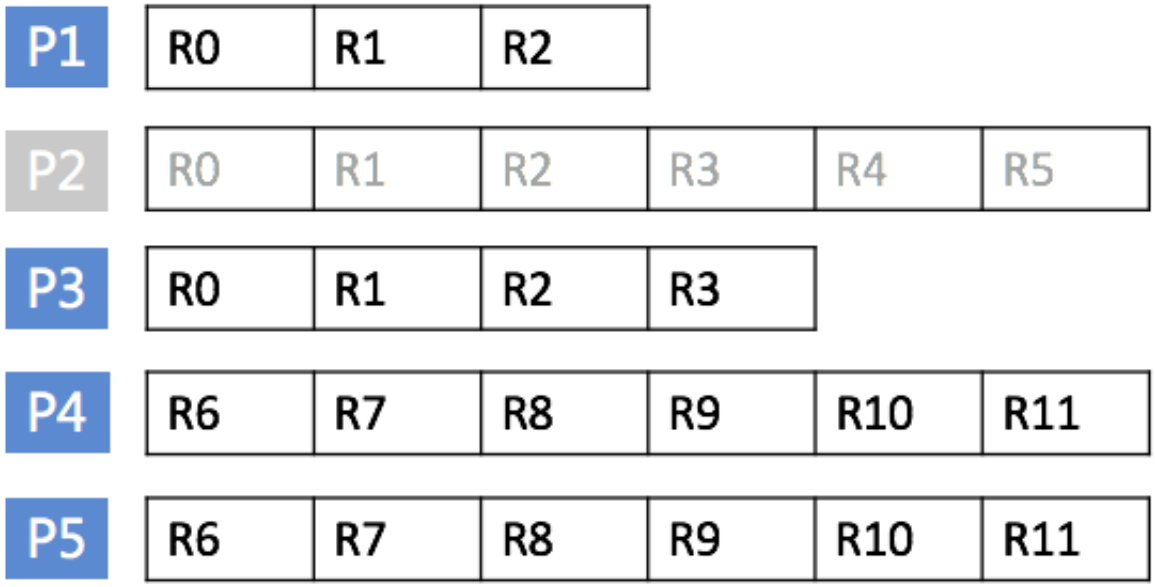
Example



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.

4.2 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 `ListStreams` 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*.

> **Note:**
>
> - 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*.

4.3 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).
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:

P1
P2
P3
P4
P5

StreamId	StatusType	StatusValue	Checkpoint	Counter	owner	Stealer	ParentShardIds
Table_123456	LeaseKey	ShardId1	checkpoint1	101	worker1		
Table_123456	LeaseKey	ShardId2	checkpoint2	95	worker2		
Table_123456	LeaseKey	ShardId3	checkpoint3	102	worker2		
Table_123456	LeaseKey	ShardId4	checkpoint4	55	worker1		p2
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 Steam 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.openservices</groupId>
  <artifactId>tablestore-streamclient</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 Steam 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
void initialize(InitializationInput initializationInput);	Used to initialize a read task. It indicates that Stream Client is about to read data of a shard.
void processRecords(ProcessRecordsInput processRecordsInput);	Indicates how the user wants to process this batch of records after the data is read. The `getCheckpointer` function in `ProcessRecordsInput` can be used to obtain `IRecordProcessorCheckpointer`. The framework provides this API to implement the checkpoint. You can determine how often the checkpoint is implemented.
void shutdown(ShutdownInput shutdownInput);	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 processRec
ordsInput) {
            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(processRecordsInput.getCheckpointer().
getLargestPermittedCheckpointValue());
            try {
                processRecordsInput.getCheckpointer().checkpoint();
            } catch (ShutdownException e) {
                e.printStackTrace();
            } catch (StreamClientException e) {
                e.printStackTrace();
            } catch (DependencyException e) {
                e.printStackTrace();
            }
        }

        public void shutdown(ShutdownInput shutdownInput) {
            // finish the query task and trace the shutdown reason
            System.out.println(shutdownInput.getShutdownReason());
        }
    }

    class RecordProcessorFactory implements IRecordProcessorFactory {

        private final String workerIdentifier;

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

        public IRecordProcessor createProcessor() {
            return new StreamSample.RecordProcessor(workerIdentifier);
        }
    }

    public Worker getNewWorker(String workerIdentifier) {
        // Please replace with your table info
        final String endPoint = "";
        final String accessId = "";
        final String accessKey = "";
        final String instanceName = "";

        StreamConfig streamConfig = new StreamConfig();
```

```
        streamConfig.setOTSClient(new SyncClient(endPoint, accessId,
accessKey,
                instanceName));
        streamConfig.setDataTableName("teststream");
        streamConfig.setStatusTableName("statusTable");

        Worker worker = new Worker(workerIdentifier, new ClientConfig
(), streamConfig,
                new StreamSample.RecordProcessorFactory(workerIden
tifier), 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();
    }
}
```

5 HBase

5.1 Table Store HBase Client

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

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

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

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

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

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

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

Items	Table Store	Self-built HBase cluster
	both the table-level and API-level.	
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.

5.2 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

Functions	Supported or Not
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

Functions	Supported or Not
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
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

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

- checkAndXXX

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

- Exist

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

- Filter

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

Functions not supported by Table Store HBase Client APIs

- Namespaces

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

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

- modifyNamespace(NamespaceDescriptor descriptor)

- Region management

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

- Snapshots

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

- Table management

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

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

- Coprocessors

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

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

- Distributed procedures

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

  - execProcedure(String signature, String instance, Map props)

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

- Increment and Append

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

5.3 Migrate from HBase to Table Store

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

Dependencies

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

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

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

```
  <dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
           <exclusions>
               <exclusion>
                   <groupId>com.aliyun.openservices</groupId>
                   <artifactId>tablestore</artifactId>
               </exclusion>
               <exclusion>
                   <groupId>org.apache.hbase</groupId>
                   <artifactId>hbase-client</artifactId>
               </exclusion>
           </exclusions>
       </dependency>
       <dependency>
           <groupId>org.apache.hbase</groupId>
           <artifactId>hbase-client</artifactId>
           <version>1.2.1</version>
       </dependency>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore</artifactId>
           <classifier>jar-with-dependencies</classifier>
           <version>4.2.0</version>
       </dependency>
```

```
    </dependencies>
```

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

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

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

Configure the file

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

- HBase Connection type

  Set Connection to TableStoreConnection.

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

- Configuration items of Table Store

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

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

```
<property>
  <name>tablestore.client.endpoint</name>
  <value></value>
 </property>
 <property>
  <name>tablestore.client.instancename</name>
  <value></value>
 </property>
 <property>
  <name>tablestore.client.accesskeyid</name>
  <value></value>
 </property>
 <property>
  <name>tablestore.client.accesskeysecret</name>
  <value></value>
 </property>
```

  — Optional items you can configure are as follows.

```
 <property>
  <name>hbase.client.tablestore.family</name>
  <value>f1</value>
```

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

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

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

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

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

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

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

- tablestore.client.retries

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

5.4 Migrate HBase of an earlier version

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

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

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

Connection APIs

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

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

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

The latest code is as follows:

```
Connection connection = ConnectionFactory.createConnection(config);
// ...
connection.close();
```

TableName series

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

The latest code is as follows:

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

Table, BufferedMutator, and RegionLocator APIs

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

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

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

Admin APIs

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

Use the Connection instance to create an Admin instance:

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

5.5 Hello World

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

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

Code position

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

Use HBase APIs

- Configure project dependencies

  Configure Maven dependencies as follows.

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

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

- Configure the file

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

  ```
  <configuration>
      <property>
          <name>hbase.client.connection.impl</name>
          <value>com.alicloud.tablestore.hbase.TablestoreConnection</
  value>
      </property>
      <property>
          <name>tablestore.client.endpoint</name>
          <value>endpoint</value>
      </property>
      <property>
          <name>tablestore.client.instancename</name>
          <value>instance_name</value>
      </property>
      <property>
          <name>tablestore.client.accesskeyid</name>
          <value>access_key_id</value>
      </property>
      <property>
          <name>tablestore.client.accesskeysecret</name>
          <value>access_key_secret</value>
      </property>
      <property>
          <name>hbase.client.tablestore.family</name>
          <value>f1</value>
      </property>
      <property>
          <name>hbase.client.tablestore.table</name>
          <value>ots_adaptor</value>
      </property>
  </configuration>
  ```

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

- Connect Table Store

Create a TableStoreConnection object to connect Table Store.

```
  Configuration config = HBaseConfiguration.create();

  // Create a Tablestore Connection
  Connection connection = ConnectionFactory.createConnection(config
);

  // Admin is used for creation, management, and deletion
        Admin admin = connection.getAdmin();
```

- Create a table

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

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

  // Create a ColumnFamily. Use the default ColumnFamily name for
Max Versions and TimeToLive. The default ColumnFamily name for Max
Versions is 1 and for TimeToLive is Integer.INF_MAX
  descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));

  // Use the createTable API of the Admin to create a table
  System.out.println("Create table " + descriptor.getNameAsString
());
  admin.createTable(descriptor);
```

- Write Data

  Write a row of data to Table Store.

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

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

  // Add a column. Table Store supports only single ColumnFamilies
. The ColumnFamily name is configured in hbase-site.xml. If the
ColumnFamily name is not configured, the default name is "f". In
this case, the value of COLUMN_FAMILY_NAME may be null when data is
written.
  put.addColumn(COLUMN_FAMILY_NAME, COLUMN_NAME, COLUMN_VALUE);

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

- Read Data

Read data of the specified row.

```
  // Create a Get object to read the row whose primary key is
ROW_KEY.
  Result getResult = table.get(new Get(ROW_KEY));
  Result result = table.get(get);

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

- Scan data

  Read data in the specified range.

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

 ResultScanner scanner = table.getScanner(scan);

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

- Delete a table

  Use Admin APIs to delete a table.

```
  print("Delete the table");
  admin.disableTable(table.getName());
  admin.deleteTable(table.getName());
```

Complete code

```
package samples;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HelloWorld {

    private static final byte[] TABLE_NAME = Bytes.toBytes("HelloTable
store");
    private static final byte[] ROW_KEY = Bytes.toBytes("row_1");
    private static final byte[] COLUMN_FAMILY_NAME = Bytes.toBytes("f
");
```

```
    private static final byte[] COLUMN_NAME = Bytes.toBytes("col_1");
    private static final byte[] COLUMN_VALUE = Bytes.toBytes("
col_value");

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

    private static void helloWorld() {

        try  {
            Configuration config = HBaseConfiguration.create();
            Connection connection = ConnectionFactory.createConnection
(config);
            Admin admin = connection.getAdmin();

            HTableDescriptor descriptor = new HTableDescriptor(
TableName.valueOf(TABLE_NAME));
            descriptor.addFamily(new HColumnDescriptor(COLUMN_FAM
ILY_NAME));

            System.out.println("Create table " + descriptor.getNameAsS
tring());
            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_VAL
UE);
            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);
```

```
        }
    }
}
```