

Alibaba Cloud Table Store

SDK Reference

Issue: 20190722

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 a 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>switch {stand slave}</code>

Contents

Legal disclaimer.....	I
Generic conventions.....	I
1 SDK overview.....	1
2 Python SDK.....	2
2.1 Preface.....	2
2.2 Installation.....	3
2.3 Table-level operations.....	4
2.3.1 Overview.....	4
2.3.2 Create a table.....	5
2.3.3 Obtain table names.....	6
2.3.4 Update a table.....	7
2.3.5 Query the description of a table.....	8
2.3.6 Delete a table.....	9
2.3.7 Create a search index.....	10
2.3.8 Obtain search indexes of a table.....	12
2.3.9 Query the description of a search index.....	12
2.3.10 Delete a search index.....	12
2.3.11 Manage global secondary indexes.....	12
2.4 Single-row operations.....	13
2.5 Multiple-row operations.....	20
2.6 Error handling.....	26
3 Java SDK.....	28
3.1 Preface.....	28
3.2 Installation.....	28
3.3 Initialization.....	29
3.4 Table-level operations.....	32
3.4.1 Overview.....	32
3.4.2 CreateTable.....	32
3.4.3 UpdateTable.....	36
3.4.4 ListTable.....	37
3.4.5 ComputeSplitsBySize.....	38
3.4.6 DescribeTable.....	38
3.4.7 DeleteTable.....	39
3.4.8 CreateIndex.....	39
3.4.9 DeleteIndex.....	40
3.4.10 CreateSearchIndex.....	41
3.4.11 DescribeSearchIndex.....	44
3.4.12 ListSearchIndex.....	44
3.4.13 DeleteSearchIndex.....	44
3.5 Tunnel Service.....	45

3.5.1 Overview.....	45
3.5.2 Quick start.....	45
3.5.3 Configure the data consumption framework.....	49
3.5.4 CreateTunnel.....	51
3.5.5 ListTunnel.....	52
3.5.6 DescribeTunnel.....	53
3.5.7 DeleteTunnel.....	55
3.6 Search Index-based operations.....	56
3.6.1 Query by exact match.....	56
3.6.2 Query by match.....	57
3.6.3 PrefixQuery.....	60
3.6.4 RangeQuery.....	61
3.6.5 WildcardQuery.....	61
3.6.6 Geographic location-based query.....	62
3.6.7 BoolQuery.....	65
3.6.8 NestedQuery.....	67
3.6.9 Sorting and paging.....	68
3.7 Single-row operations.....	70
3.8 Condition update.....	77
3.9 Filter.....	79
3.10 Primary key column auto-increment.....	81
3.11 Error handling.....	83
3.12 Incremental data operations.....	84
3.13 Multiple-row operations.....	86
4 Go SDK.....	93
4.1 Preface.....	93
4.2 Installation.....	93
4.3 Initialization.....	94
4.4 Single-row operations.....	95
4.5 Multiple-row operations.....	98
4.6 Tunnel Service-level operations.....	102
4.6.1 Installation.....	102
4.6.2 Quick start.....	102
4.6.3 Configuration items.....	104
4.6.4 Troubleshooting.....	106
4.7 Table-level operations.....	108
4.7.1 Create tables.....	108
4.7.2 List table names.....	110
4.7.3 Update tables.....	110
4.7.4 Query the description information of a table.....	111
4.7.5 Delete tables.....	112
4.7.6 Create search indexes.....	112
4.7.7 List search indexes.....	115
4.7.8 Query descriptive information of search indexes.....	115
4.7.9 Delete search indexes.....	116

4.8 Search index-based queries.....	116
4.8.1 Query by exact match.....	116
4.8.2 Query by match.....	118
4.8.3 Query by prefix.....	121
4.8.4 Query by range.....	122
4.8.5 Query by wildcard character.....	123
4.8.6 Query by geographic location.....	124
4.8.7 Query by combination of conditions.....	126
4.8.8 Query by nested field.....	129
4.8.9 Index sorting and page scanning.....	130
5 NodeJS SDK.....	134
5.1 Preface.....	134
5.2 Installation.....	134
5.3 Initialization.....	135
5.4 Long type.....	136
5.5 Table-level operations.....	137
5.5.1 Create tables.....	137
5.5.2 List table names.....	138
5.5.3 Update tables.....	139
5.5.4 Query the description information of a table.....	139
5.5.5 Delete tables.....	140
5.5.6 Create search indexes.....	141
5.5.7 List search indexes.....	144
5.5.8 Query descriptive information of search indexes.....	144
5.5.9 Delete search indexes.....	144
5.6 Single-row operations.....	145
5.7 Multiple-row operations.....	148
5.8 Search index-based queries.....	152
5.8.1 Query by exact match.....	153
5.8.2 Query by match.....	155
5.8.3 Query by prefix.....	157
5.8.4 Query by range.....	158
5.8.5 Query by wildcard character.....	159
5.8.6 Query by geographic location.....	160
5.8.7 Query by combination of conditions.....	163
5.8.8 Query by nested field.....	165
5.8.9 Index sorting and page scanning.....	166
5.9 Error handling.....	170
6 .NET SDK.....	171
6.1 Preface.....	171
6.2 Initialization.....	172
6.3 Table operations.....	174
6.3.1 CreateTable.....	174
6.3.2 ListTable.....	175

6.3.3 UpdateTable.....	176
6.3.4 DescribeTable.....	177
6.3.5 DeleteTable.....	178
6.3.6 CreateSearchIndex.....	179
6.3.7 DescribeSearchIndex.....	181
6.3.8 ListSearchIndex.....	182
6.3.9 DeleteSearchIndex.....	182
6.4 Searchindex operations.....	183
6.4.1 TermQuery.....	183
6.4.2 Query by match.....	183
6.4.3 PrefixQuery.....	185
6.4.4 RangeQuery.....	186
6.4.5 WildcardQuery.....	187
6.4.6 Query by geographic location.....	187
6.4.7 BoolQuery.....	190
6.4.8 NestedQuery.....	190
6.4.9 Sorting and paging.....	191
6.5 Single-row operations.....	194
6.6 Multiple-row operations.....	203
6.7 Error handling.....	209
7 C++ SDK.....	210
7.1 Preface.....	210
7.2 Installation.....	210
7.3 Initialization.....	214
7.4 Table-level operations.....	217
7.5 Single-row operations.....	221
7.6 Multiple-row operations.....	224
7.7 Filter.....	229
7.8 Asynchronous interface.....	231
7.9 Logs.....	233
7.10 Error handling.....	235
7.11 Retry.....	236
8 PHP SDK.....	238
8.1 Overview.....	238
8.2 Installation.....	239
8.3 Initialization.....	244
8.4 Table operations.....	247
8.4.1 Create a table.....	247
8.4.2 Obtain table names.....	252
8.4.3 Update a table.....	253
8.4.4 Delete a table.....	258
8.4.5 Split data with a specified size.....	259
8.4.6 Query the description of a table.....	261
8.4.7 Create a search index.....	265

8.4.8 Obtain search indexes of a table.....	269
8.4.9 Query the description of a search index.....	270
8.4.10 Delete a search index.....	270
8.4.11 Create a global secondary index.....	271
8.4.12 Delete a global secondary index.....	273
8.5 Table operations.....	273
8.6 Local transactions.....	291
8.7 Single-row operations.....	293
8.8 Multi-row operations.....	311
8.9 Search index.....	332
8.9.1 Query by exact match.....	336
8.9.2 Query by match.....	337
8.9.3 Query by prefix.....	340
8.9.4 Query by range.....	340
8.9.5 Query by wildcard character.....	341
8.9.6 Query by geographic location.....	342
8.9.7 Query by column existence.....	345
8.9.8 Query by a combination of filtering conditions.....	346
8.9.9 Query by nested field.....	350
8.9.10 Sorting and paging.....	351
8.10 Conditional update.....	355
8.11 Filter.....	361
8.12 Auto-increment primary key column.....	365
8.13 Error handling.....	368
8.14 FAQ.....	368
9 Download SDK.....	370
9.1 Java SDK.....	370
9.2 NodeJS SDK.....	372
9.3 Python SDK.....	373
9.4 .Net SDK.....	375
9.5 PHP SDK.....	377

1 SDK overview

Before using the table store SDK, you must:

- Activate [Alibaba Cloud Table Store Service](#).
- [Create an AccessKey](#).

This document describes how to install and use Java SDK v4.0.0 and later for Table Store (formerly OTS).

Language	References
Java	Java SDK
Python	Python SDK
C++	C++ SDK
Go	Go SDK
.NET	.Net SDK
NodeJS	NodeJS SDK

2 Python SDK

2.1 Preface

This document describes how to install and use Python SDK v4.x.x for Table Store . Make sure that you have activated Alibaba Cloud Table Store and created an AccessKeyID and AccessKeySecret.

- If you have not activated Alibaba Cloud Table Store or want to learn more about this service, visit the [Table Store product homepage](#).
- If you have not created an AccessKeyID and AccessKeySecret, log on to the [AccessKey console](#) to create an AccessKey.

Download the SDK

- [SDK](#)
- [GitHub](#)

For more information about version history, [click here](#).

Compatibility

- Table Store is compatible with Python SDK v5.x.x.
- Table Store is compatible with Python SDK v4.x.x.

- Table Store is incompatible with Python SDK v2.x.x, because Python SDK v2.x.x supports out-of-order primary keys. Detailed incompatibilities include:
 - The name of the SDK is changed from ots2 to tablestore.
 - The `Client.create_table` operation is added with the `TableOptions` parameter.
 - The `primary_key` parameter for the `put_row`, `get_row`, and `update_row` operations is changed from the dict type to the list type to guarantee sequence.
 - The `attribute_columns` parameter for the `put_row` and `update_row` operations is changed from the dict type to the list type.
 - The `attribute_columns` parameter for the `put_row` and `update_row` operations is added with `timestamp`.
 - The `get_row` and `get_range` operations are added with the `max_version` and `time_range` parameters. At least one parameter must exist.
 - The `put_row`, `update_row`, and `delete_row` operations are added with the `return_type` parameter. Currently, only `RT_PK` is supported, indicating that the returned value contains the PK value of the current row.
 - The values returned by the `put_row`, `update_row`, and `delete_row` operations are added with `return_row`. If `return_type` is specified as `RT_PK` in a request, `return_row` contains the PK value of the current row.

Version

Latest version: 5.1.0

2.2 Installation

Prerequisites

Applicable to Python 2 and Python 3.

Installation

- Method 1: Install Python by using pip.

The command is as follows.

```
sudo pip install tablestore
```

- Method 2: Install Python by using GitHub.

Make sure you have installed [git](#), and then run the following command.

```
git clone https://github.com/aliyun/tablestore-python-sdk.git
sudo python setup.py install
```

- Method 3: Install Python by using source code.

1. Download the [Python SDK](#).
2. Decompress the SDK, and then run the following command.

```
sudo python setup.py install
```

Verify SDK

Enter python on the command line and press ENTER to check the version.

```
>>> import tablestore
>>> tablestore.__version__
'5.1.0'
```

Uninstall SDK

Uninstall python SDK by using pip.

```
sudo pip uninstall tablestore
```

2.3 Table-level operations

2.3.1 Overview

The Python SDK of Table Store allows you to perform a variety of table operations, including:

[Create a table](#)

[Obtain table names](#)

[Update a table](#)

[Query the description of a table](#)

[Delete a table](#)

[Create a search index](#)

[Obtain search indexes of a table](#)

[Query the description of a search index](#)

[Delete a search index](#)

[Manage global secondary indexes](#)

2.3.2 Create a table

You can call the CreateTable operation to create a table based on the specified table schema information.

When creating a table in Table Store, you must specify the primary key of the table. A primary key contains one to four primary key columns. Each primary key column has a name and a type.

Method

```

    Descriptio n : You can call this method to
    create a table based on the specified table schema
    informatio n .

    `` table_meta `` is an instance of the `` tablestore
    . metadata . TableMeta `` class . It contains the table
    name and primary key schema .
    For more informatio n , see the documentat ion
    for the `` TableMeta `` class . After you create a table
    , it takes about 1 minute to load the partitions .
    After that , you can perform table operations .
    `` table_opti ons `` is an instance of the ``
    tablestore . metadata . TableOptio ns `` class . It contains
    the time_to_li ve , max_versio n , and max_time_d eviation
    parameters .
    `` reserved_t hroughput `` is an instance of the
    `` tablestore . metadata . ReservedTh roughput `` class . It
    specifies the reserved read throughput and reserved
    write throughput .

    Return value : none .

    def create_tab le ( self , table_meta , reserved_t hroughput
    ) :

```



Note:

After the table is created in Table Store, it takes several seconds to load the table. Do not perform any operations during this period.

Example

The following code provides an example of how to create a table. In this example, a table with two primary key columns is created. Specifically, `time_to_live` is set to 31,536,000 seconds (one year), `max_version` to 3, `max_time_deviation` to 86,400 seconds (one day), and `reserved_throughput` to (0,0).

```
# Create a schema for the primary key columns ,
including the quantity , names , and types of primary
keys .
# The first primary key column ( partition column ) is
named pk0 and of the integer type .
# The second primary key column is named pk1 and
of the integer type . You can also set the column
to the string or binary type .
schema_of_primary_key = [(' pk0 ', ' INTEGER '), (' pk1 ', '
INTEGER ')]

# Create a tableMeta object based on the table name
and the schema of primary key columns .
table_meta = TableMeta ( ' SampleTable ', schema_of_
primary_key )

# Create TableOptions . Set time_to_live to 31 , 536
, 000 seconds ( data is cleared automatica lly when
expired ) , max_versio n to 3 , and max_time_d eviation to
86 , 400 seconds ( one day ) .
table_opti ons = TableOptio ns ( 31536000 , 3 , 86400 )

# Set both the reserved read throughput and reserved
write throughput to 0 .
reserved_t hroughput = ReservedTh roughput ( CapacityUn it ( 0
, 0 ))

# Call the create_tab le method of the client . If
no exception is thrown , the operation is successful .
If an exception is thrown , the operation fails .
try :
    ots_client . create_tab le ( table_meta , table_opti ons ,
reserved_t hroughput )
    print " create table succeeded "
# Handle any exception that is thrown .
except Exception :
    print " create table failed ."
```

You can obtain the full sample code at [CreateTable@GitHub](#).

2.3.3 Obtain table names

You can call the `ListTable` operation to obtain the names of all tables that are created in the current instance.

Method

```
"""
    Descriptio n : You can call this method to
    obtain all table names .
```

```

        Return value : the table name list .
        ``table_list`` indicates the table name list of
        the tuple type . For example : (' MyTable1 ', ' MyTable2 ').
    """
    def list_table ( self ):
```

Example

The following code provides an example of how to obtain the names of all tables in an instance.

```

try :
    list_response = ots_client . list_table ()
    print ' table list : '
    for table_name in list_response :
        print table_name
    print " list table succeeded "
except Exception :
    print " list table failed ."
```

You can obtain the full sample code at [ListTable@GitHub](#).

2.3.4 Update a table

You can call the UpdateTable operation to update the reserved read throughput or reserved write throughput of a specified table.

Method

```

    """
        Descriptio n : You can call this method to
        update attributes of a table . This method can only
        modify the reserved read throughput or reserved write
        throughput currently .
        ``table_name`` specifies the name of the table .
        ``table_opti ons`` is an instance of the ``tablestore
        . metadata . TableOptio ns`` class . It contains the
        time_to_li ve , max_versio n , and max_time_d eviation
        parameters .
        ``reserved_t hroughput`` is an instance of the ``
        ots2 . metadata . ReservedTh roughput`` class . It specifies
        the reserved read throughput and reserved write
        throughput .

        Return value : the time when the reserved read
        throughput or reserved write throughput was raised or
        lowered most recently for this table and the number
        of times it was lowered today .

        ``update_tab le_respons e`` indicates the update
        result . It is an instance of the ots2 . metadata .
        UpdateTabl eResponse class .
    """
    def update_tab le ( self , table_name , table_opti ons ,
        reserved_t hroughput ):
```

Example

The following code provides an example of how to update the reserved read throughput or reserved write throughput of a table.

```
# Set both the reserved read throughput and
reserved write throughput to 0 .
reserved_t hroughput = ReservedTh roughput ( CapacityUn
it ( 0 , 0 ))

# Create TableOptio ns . Set time_to_li ve to 31 , 536
, 000 seconds ( data is cleared automatica lly when
expired ), max_versio n to 5 , and max_time_d eviation to
one day 86 , 400 seconds ( one day ).
table_opti ons = TableOptio ns ( 31536000 , 5 , 86400 )

try :
    # Call the update_tab le method to update
the reserved read throughput and reserved write
throughput of the table .
    ots_client . update_tab le ( ' SampleTabl e ' ,
reserved_t hroughput )

    # If no exception is thrown , the operation
is successful .
    print " update table succeeded "
except Exception :
    # If an exception is thrown , the operation
fails . Handle the exception .
    print " update table failed "
```



Note:

You can obtain the full sample code at [updateTable@GitHub](#).

2.3.5 Query the description of a table

You can call the DescribeTable operation to query the schema information, reserved read throughput, and reserved write throughput of a table.

Method

```
"""
    Descriptio n : You can call this method to
obtain the descriptio n of a table .
    `` table_name `` specifies the name of the table .
    Return value : the descriptio n of the table .
    `` describe_t able_respo nse `` indicates the
descriptio n of the table , which is an instance of
the ots2 . metadata . DescribeTa bleRespons e class .
"""
def describe_t able ( self , table_name ):
```

Example

The following code provides an example of how to obtain the description of a table.

```
try :
    describe_r esponse = ots_client . describe_t able ( ' myTable ' )
```

```
# If no exception is thrown, the operation is
successful. Display the table information.
print "describe table succeeded."
print ('TableName : % s ' % describe_r esponse . table_meta .
table_name )
print (' PrimaryKey : % s ' % describe_r esponse . table_meta .
schema_of_ primary_ke y )
print (' Reserved read throughput : % s ' % describe_r
esponse . reserved_t hroughput_ details . capacity_u nit . read )
print (' Reserved write throughput : % s ' % describe_r
esponse . reserved_t hroughput_ details . capacity_u nit . write )
print (' Last increase throughput time : % s ' %
describe_r esponse . reserved_t hroughput_ details . last_incre
ase_time )
print (' Last decrease throughput time : % s ' %
describe_r esponse . reserved_t hroughput_ details . last_decre
ase_time )
print (' table options \' s time to live : % s ' %
describe_r esponse . table_opti ons . time_to_li ve )
print (' table options \' s max version : % s ' %
describe_r esponse . table_opti ons . max_versio n )
print (' table options \' s max_time_d eviation : % s ' %
describe_r esponse . table_opti ons . max_time_d eviation )
except Exception :
# If an exception is thrown, the operation fails .
Handle the exception .
print "describe table failed ."
```

**Note:**

You can obtain the full sample code at [describeTable@GitHub](#).

2.3.6 Delete a table

You can call the DeleteTable operation to delete a table from the current instance.

Method

```
"""
    Descriptio n : You can call this method to
delete a table based on the table name .
    ` table_name ` specifies the name of the table .
    Return value : none .
"""
def delete_tab le ( self , table_name ):
```

Example

The following code provides an example of how to delete a table.

```
try :
# Call the delete_tab le method to delete the table
named SampleTabl e .
ots_client . delete_tab le (' SampleTabl e ')
# If no exception is thrown, the operation is
successful
print "delete table succeeded "
# If an exception is thrown, the operation fails .
Handle the exception .
```

```
except Exception :  
    print " delete table failed "
```

**Note:**

You can obtain the full sample code at [deleteTable@GitHub](#).

2.3.7 Create a search index

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

If a table needs to support complex queries, you must create a search index for the table. When creating a search index, you must specify the `table_name`, `index_name`, and `index_meta` parameters.

- `table_name`: the name of the table for which the search index is created.
- `index_name`: the name of the search index to be created.
- `index_meta`: the index schema, including the `index_setting`, `field_schemas`, and `index_sort` parameters.
 - `field_schemas`: the list of `FieldSchemas`. Each `FieldSchema` contains the following parameters:
 - `field_name` (required, String): the name of the field to be indexed, that is, the column name. The field can be a primary key column or an attribute column.
 - `field_type` (required, `FieldType`): the type of the field. For more information, see the description of field types for a search index.
 - `is_array` (optional, Boolean): specifies whether the column is an array. A value of true indicates that the column is an array. Data written to the column must be a JSON array, such as `["a","b","c"]`. You do not need to explicitly specify this parameter for nested columns because they are arrays.
 - `index` (optional, Boolean): specifies whether to index the column.
 - `analyzer` (optional, `AnalyzerType`): the tokenizer. You can specify this parameter if the column is a text column.
 - `enable_sort_and_agg` (optional, Boolean): specifies whether to enable sorting and statistics collection for the column.
 - `store` (optional, Boolean): specifies whether to store the values of the field in the search index. A value of true indicates that you can read the values of the

field directly from the search index without the need to query the table. This optimizes query performance.

- **sub_field_schemas** (optional, FieldSchema list): the list of FieldSchemas for sub-fields. If the column is a nested column, you must specify this parameter to configure the index types of sub-columns in the nested column.

- **index_setting** (optional):

routing_fields (optional, advanced feature): the custom routing fields. You can specify some primary key columns as routing fields. Table Store distributes data that is written to a search index to different partitions based on the specified routing fields. The data with the same routing field values is distributed to the same partition.

- **index_sort** (optional): specifies how the index is sorted. The index is sorted in the same way as the primary key of the table by default. (If the index contains nested fields, it is not sorted by default).

- **sorters** (required, Sort array): the sorting methods. Valid values:

- **PrimaryKeySort**: The index is sorted by the primary key in ascending or descending order.

- **FieldSort**: The index is sorted by a specified column in ascending or descending order.

```
field_a = FieldSchema('k', FieldType.KEYWORD, index =
True, enable_sort_and_agg = True, store = True)
field_b = FieldSchema('t', FieldType.TEXT, index = True
, store = True, analyzer = AnalyzerType.SINGLEWORD)
field_c = FieldSchema('g', FieldType.GEOPPOINT, index =
True, store = True)
field_d = FieldSchema('ka', FieldType.KEYWORD, index =
True, is_array = True, store = True)
field_e = FieldSchema('la', FieldType.LONG, index = True
, is_array = True, store = True)

field_n = FieldSchema('n', FieldType.NESTED, sub_field_
schemas=[
    FieldSchema('nk', FieldType.KEYWORD, index = True,
store = True),
    FieldSchema('nl', FieldType.LONG, index = True, store
= True),
    FieldSchema('nt', FieldType.TEXT, index = True, store
= True),
])

fields = [field_a, field_b, field_c, field_d, field_e,
field_n]

index_setting = IndexSetting(routing_fields=['PK1'])
```

```
index_sort = None # The index cannot be sorted if
there is any nested field .
# index_sort = Sort ( sorters =[ PrimaryKey Sort ( SortOrder . ASC
)])
index_meta = SearchIndexMeta ( fields , index_setting =
index_setting , index_sort = index_sort )
client . create_search_index ( table_name , index_name ,
index_meta )
```

2.3.8 Obtain search indexes of a table

You can call the ListSearchIndex operation to obtain all search indexes created for a table.

Sample code:

```
for table , index_name in client . list_search_index (
table_name ):
    print table , index_name
```

2.3.9 Query the description of a search index

You can call the DescribeSearchIndex operation to query the description of a search index for a table, including fields and configurations of the search index.

Sample code:

```
index_meta , sync_stat = client . describe_search_index (
table_name , index_name )
print json . dumps ( index_meta , default = lambda x : x .
__dict__ , indent = 2 )
print json . dumps ( sync_stat , default = lambda x : x .
__dict__ , indent = 2 )
```

2.3.10 Delete a search index

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

Sample code:

```
client . delete_search_index ( table_name , index_name )
```

2.3.11 Manage global secondary indexes

Create a global secondary index

You can create a global secondary index when creating a table or add a global secondary index to an existing table. You must use `defined_columns` to specify the fields that can be used by a global secondary index when creating a table.

Create a global secondary index when creating a table

```
schema_of_primary_key = [('gid', 'INTEGER'), ('uid', 'STRING')]
defined_columns = [('i', 'INTEGER'), ('bool', 'BOOLEAN'), ('d', 'DOUBLE'), ('s', 'STRING'), ('b', 'BINARY')]
table_meta = TableMeta(table_name, schema_of_primary_key, defined_columns)
table_option = TableOptions(-1, 1)
reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
secondary_indexes = [
    SecondaryIndexMeta('index1', ['i', 's'], ['gid', 'uid', 'bool', 'b', 'd']),
]
client.create_table(table_meta, table_option, reserved_throughput, secondary_indexes)
```

Add a global secondary index to an existing table

```
index_meta = SecondaryIndexMeta('index2', ['i', 's'], ['gid', 'uid', 'bool', 'b', 'd'])
client.create_secondary_index(table_name, index_meta)
```

Delete a global secondary index

```
client.delete_secondary_index(table_name, 'index1')
```

2.4 Single-row operations

The Table Store SDK provides the following single-row operation APIs: PutRow, GetRow, UpdateRow, and DeleteRow.

PutRow

Inserts data into a specified row.

API

```
"""
    Description: This operation writes a single row of data. CapacityUnit consumed by the operation is returned.
    ``table_name`` is the name of the table.
    ``row`` indicates row data, including primary key columns and attribute columns.
    ``condition`` indicates the conditions to be checked before an operation is executed. If the conditions are met, the operation is executed. It is an instance of the tablestore.metadata.Condition class. Currently, this function only supports row existence checks. Check conditions include: 'IGNORE', 'EXPECT_EXIST', and 'EXPECT_NOT_EXIST'.
    ``return_type`` indicates the type of the returned value. It is an instance of the tablestore

```

```
. metadata . ReturnType class . Currently , only return of
PrimaryKey is supported , which is typically used in
automatic increment of primary key columns .

    Return : CapacityUn its consumed by the operation
, , and the row data to be returned .
    consumed `` indicates the CapacityUn its consumed . It
is an instance of the tablestore . metadata . CapacityUn
it class .
    `` return_row `` indicates the row data returned
, which may include the primary key and attribute
columns .
    """
    def put_row ( self , table_name , row , condition = None ,
return_type = None )
```

Examples

Insert a data row.

```
## The first primary key column of the primary key
is gid with an integral value of 1 . The second
primary key column is uid with an integral value of
101 .
primary_key = [(' gid ', 1 ), (' uid ', 101 )]

## Four attribute columns exist :
## The first attribute column is "
name " with the string value " John " . The version is
not specified , and the current system time is used
as the version .
## The second attribute column is
" mobile " with the integral value 1510000000 0 . The
version is not specified , and the current system time
is used as the version .
## The third attribute column is "
address " with the binary value " China " . The version
is not specified , and the current system time is
used as the version .
## The fourth column is " age " with
the value 29 . 7 . The version is 1498184687 .
attribute_columns = [(' name ', ' John '), (' mobile ',
1510000000 0 ), (' address ', bytearray ( ' China ' )), (' female ',
False ), (' age ', 29 . 7 , 1498184687 000 )]

## Constructi on of Row using primary_key and
attribute_columns
row = Row ( primary_key , attribute_columns )

# Row condition : The expected row does not exist
. If the expected row exists , the error " Condition
Update Failed " is returned .
condition = Condition ( RowExisten ceExpectat ion .
EXPECT_NOT _EXIST )

try :
# Call the put_row method . If ReturnType is not
specified , return_row is None .
consumed , return_row = client . put_row ( table_name , row
, condition )
```

```
# The write CU consumed by this request is printed
.
print ('put row succeed , consume %s write cu .' %
consumed . write )
# The client is abnormal , which is typically due to
incorrect parameters or a network error .
except OTSClientError as e :
    print "put row failed , http_status :%d , error_message :%s " % ( e . get_http_status () , e . get_error_message () )
# The server is abnormal , which is typically due to
incorrect parameters or a network error .
except OTSServiceError as e :
    print "put row failed , http_status :%d , error_code :%s , error_message :%s , request_id :%s " % ( e . get_http_status () , e . get_error_code () , e . get_error_message () , e . get_request_id () )
```



Note:

- RowExistenceExpectation.IGNORE indicates that new data is inserted no matter whether the specified row exists or not. If the inserted data is the same as the existing data, the existing data is overwritten.
- RowExistenceExpectation.EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten.
- RowExistenceExpectation.EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist.
- In the preceding sample program, the version of the attribute column “age” is 1498184687000, and the value is June, 23, 2017. If the current time minus the value of max_time_deviation (specified during table creation) is greater than 1498184687000, the PutRow operation is disabled.
- If the operation is successful, no exception is thrown; if the operation fails, an exception is thrown.

Obtain code details at [PutRow@GitHub](#).

GetRow

Reads a single data row based on a given Primary Key.

API

```
"""
    Descriptio n : This operation reads a single row
of data .
    `` table_name `` is the name of the table .
    `` primary_key `` indicates the primary key ; type
: dict .
    `` columns_to_get `` is an optional parameter that
indicates the names of the columns to read in
list format . If not entered , all columns are read .
"""
```

```

    ``column_filter`` is an optional parameter ,
    which indicates reading the row that meets specified
    conditions .
    ``max_version`` is an optional parameter , which
    indicates the maximum number of read versions .
    ``time_range`` is an optional parameter , which
    indicates reading versions within the specified range
    or reading the specified version . time_range and
    max_version are mutually exclusive .

    Return : The CapacityUnits consumed by this
    operation , the primary key columns , and attribute
    columns .
    ``consumed`` indicates the CapacityUnits consumed
    . It is an instance of the tablestore . metadata .
    CapacityUnit class .
    ``return_row`` indicates row data , including
    primary key columns and attribute columns of the list
    type , for example , [(' PK0 ', value0 ), (' PK1 ', value1 )].
    ``next_token`` indicates the location of the next
    read operation in the case of wide row reading .
    The value is encoded in binary format .
    """
    def get_row ( self , table_name , primary_key , columns_to
_get = None ,
                  column_filter = None , max_version = None ,
time_range = None ,
                  start_column = None , end_column = None , token =
None ) :

```

Examples

Read a data row.

```

# The first column of the primary key is uid with
the integral value 1 . The second column is gid
with the integral value 101 .
primary_key = [(' uid ', 1 ), (' gid ', 101 )]

# Attribute columns to be returned : name , growth , and
type If columns_to_get is [], all attribute columns
are returned .
columns_to_get = [' name ', ' growth ', ' type ' ]

# Add column filter : This row is returned if the
value of the growth column is not 0.9 .
cond = CompositeCondition ( LogicalOperator . AND )
cond . add_sub_condition ( SingleColumnCondition ( " growth
", 0.9 , Comparator Type . NOT_EQUAL ))
cond . add_sub_condition ( SingleColumnCondition ( " name ",
' Hangzhou ', Comparator Type . EQUAL ))

try :
    # Call the get_row query interface . The last
    parameter 1 indicates that the value of only one
    version needs to be returned .
    consumed , return_row , next_token = client . get_row (
table_name , primary_key , columns_to_get , cond , 1 )
    print ( ' Read succeed , consume %s read cu .' %
consumed . read )
    print ( ' Value of primary key : %s ' % return_row .
primary_key )

```

```

    print (' Value of attribute : % s ' % return_row .
attribute_ columns )
    for att in return_row . attribute_ columns :
        # Print the key , value , and version of each column
        .
        print (' name :% s \ tvalue :% s \ ttimestamp :% d ' % (
att [ 0 ], att [ 1 ], att [ 2 ]))
    # The client is abnormal , which is typically due to
    incorrect parameters or a network error .
    except OTSClientE rror as e :
        print " get row failed , http_statu s :% d , error_mess
age :% s " % ( e . get_http_s tatus (), e . get_error_ message ())
    # The server is abnormal , which is typically due to
    incorrect parameters or a network error .
    except OTSService Error as e :
        print " get row failed , http_statu s :% d , error_code :
% s , error_mess age :% s , request_id :% s " % ( e . get_http_s
tatus (), e . get_error_ code (), e . get_error_ message (), e .
get_reques t_id ())

```

**Note:**

If you query a data row, the system returns the data in all columns of the row. You can use the `columns_to_get` parameter to read the data in specified columns. For example, the system only returns the data in `col0` and `col1` if `col0` and `col1` are inserted into `columns_to_get`.

Obtain code details at [GetRow@GitHub](#).

UpdateRow

Updates the data of the specified row. If the row does not exist, a new row is added. If the row exists, the values of the specified columns are added, modified, or deleted based on the request content.

API

```

"""
    Descriptio n : This operation updates a single
row of data .
    `` table_name `` is the name of the table .
    `` row `` indicates the updated row data , including
primary key columns ( of the list type ) and
attribute columns ( of the dict type ).
    `` condition `` indicates the conditions to be
checked before an operation is executed . If the
conditions are met , the operation is executed . It
is an instance of the tablestore . metadata . Condition
class .
    Currently , this function only supports row
existence checks . Check conditions include : ' IGNORE ' , '
EXPECT_EXI ST ' , and ' EXPECT_NOT _EXIST ' .
    `` return_ty pe `` indicates the type of the
returned value . It is an instance of the tablestore
. metadata . ReturnTy pe class . Currently , only return of

```

```

PrimaryKey is supported, which is typically used in
automatic increment of primary key columns.
Return: CapacityUn its consumed by the operation
, and the row data to be returned (which is
indicated by return_row).
consumed indicates the CapacityUn its consumed. It
is an instance of the tablestore.metadata.CapacityUn
it class.
return_row indicates the row data to be
returned.
"""
def update_row ( self , table_name , row , condition ,
return_type = None )

```

Examples

Update a data row.

```

# The first column of the primary key is uid with
the integral value 1. The second column is gid
with the integral value 101.
primary_key = [('uid', 1), ('gid', 101)]

# The update contains three parts: PUT, DELETE, and
DELETE_ALL.
# PUT: Add two columns. The first column is "name"
and the value is David; the second column is "
address" and the value is Hongkong.
# DELETE: Delete the value of the address column
with the version 1488436949 003.
# DELETE_ALL: Delete the values of all versions in
the "mobile" and "age" columns.
update_of_attribute_columns = {
    'PUT': [('name', 'David'), ('address', 'Hongkong')],
    'DELETE': [('address', None, 1488436949 003)],
    'DELETE_ALL': [('mobile'), ('age')],
}
row = Row ( primary_key , update_of_attribute_columns )

# The row condition is Ignore, indicating that data
is updated regardless of whether the specified row
exists.
condition = Condition ( RowExistenceExpectation.IGNORE ,
SingleColumnCondition ("age", 20, ComparatorType.EQUAL
)) # update row on \
ly when this row is exist

try :
    consumed, return_row = client.update_row ( table_name ,
row , condition )
# The client is abnormal, which is typically due to
incorrect parameters or a network error.
except OTSClientError as e :
    print "update row failed, http_status:%d, error_mess
age:%s" % ( e.get_http_status(), e.get_error_message())
# The server is abnormal, which is typically due to
incorrect parameters or a network error.
except OTSServiceError as e :
    print "update row failed, http_status:%d, error_code
:%s, error_message:%s, request_id:%s" % ( e.get_http_s

```

```
tatus (), e . get_error_ code (), e . get_error_ message (), e .
get_reques t_id ())
```

Obtain code details at [UpdateRow@GitHub](#).

DeleteRow

API

```
"""
    Descriptio n : This operation deletes a single
    row of data .
    `` table_name `` is the name of the table .
    `` row `` indicates row data , which contains only
    the primary key in the case of delete_row .
    `` condition `` indicates the conditions to be
    checked before an operation is executed . If the
    conditions are met , the operation is executed . It
    is an instance of the tablestore . metadata . Condition
    class .
    Currently , this function only supports row
    existence checks . Check conditions include : ' IGNORE ', '
    EXPECT_EXI ST ', and ' EXPECT_NOT _EXIST '.
    Return : CapacityUn its consumed by the operation
    , and the row data to be returned ( which is
    indicated by return_row ).
    consumed indicates the CapacityUn its consumed . It
    is an instance of the tablestore . metadata . CapacityUn
    it class .
    return_row indicates the row data to be
    returned .
    """
    def delete_row ( self , table_name , row , condition ,
    return_typ e = None ):
```

Examples

Delete a data row.

```
primary_ke y = [(' gid ', 1 ), (' uid ',' 101 ')]
row = Row ( primary_ke y )
try :
    consumed , return_row = client . delete_row ( table_name ,
    row , None )
# The client is abnormal , which is typically due to
# incorrect parameters or a network error .
except OTSClientE rror as e :
    print " update row failed , http_statu s :% d , error_mess
age :% s " % ( e . get_http_s tatus (), e . get_error_ message ())
# The server is abnormal , which is typically due to
# incorrect parameters or a network error .
except OTSService E rror as e :
    print " update row failed , http_statu s :% d , error_code
:% s , error_mess age :% s , request_id :% s " % ( e . get_http_s
tatus (), e . get_error_ code (), e . get_error_ message (), e .
get_reques t_id ())
    print (' Delete succeed , consume % s write cu .' %
consumed . write )
```

Obtain code details at [DeleteRow@GitHub](#).

2.5 Multiple-row operations

The Table Store SDK provides the following multi-row operation APIs: `BatchGetRow`, `BatchWriteRow`, and `GetRange`.

BatchGetRow

Reads several data rows in batches from one or more tables.

The `BatchGetRow` operation is essentially a set of multiple `GetRow` operations.

Each operation is executed, results are returned, and capacity units are consumed independently.

Compared to the execution of a large number of `GetRow` operations, the `BatchGetRow` operation effectively reduces the request response time and increases the data read rate.

API

```

"""
    Descriptio n : This operation batch reads data
    from multiple rows .
    request = BatchGetRo wRequest ()
    request . add ( TableInBat chGetRowIt em ( myTable0 ,
primary_ke ys , column_to_ get = None , column_fil ter = None ))
    request . add ( TableInBat chGetRowIt em ( myTable1 ,
primary_ke ys , column_to_ get = None , column_fil ter = None ))
    request . add ( TableInBat chGetRowIt em ( myTable2 ,
primary_ke ys , column_to_ get = None , column_fil ter = None ))
    request . add ( TableInBat chGetRowIt em ( myTable3 ,
primary_ke ys , column_to_ get = None , column_fil ter = None ))
    response = client . batch_get_ row ( request )
    `` response `` indicates the returned results of
the tablestore . metadata . BatchGetRo wResponse type .
"""
def batch_get_ row ( self , request ):
```

Examples

In this example, 3 rows of data is batch read.

```

# Columns to be returned
columns_to_ get = [ ' name ' , ' mobile ' , ' address ' , ' age ' ]

# Read three rows
rows_to_ge t = []
for i in range ( 0 , 3 ):
    primary_ke y = [ ( ' gid ' , i ) , ( ' uid ' , i + 1 ) ]
    rows_to_ge t . append ( primary_ke y )

# Filter conditions : name is " Johb " and address is "
China ".
cond = CompositeC olumnCondi tion ( LogicalOpe rator . AND )
```

```

    cond . add_sub_co ndition ( SingleColu mnConditio n (" name ",
    " John ", Comparator Type . EQUAL ))
    cond . add_sub_co ndition ( SingleColu mnConditio n (" address
    ", ' China ', Comparator Type . EQUAL ))

# Construct a batch read request .
request = BatchGetRo wRequest ()

# Add table : Rows to be read from the table
indicated by table_name . The last parameter 1 indicates
reading the latest version .
request . add ( TableInBat chGetRowIt em ( table_name ,
rows_to_ge t , columns_to _get , cond , 1 ))

# Add table : Rows to be read from the table
indicated by notExistTa ble .
request . add ( TableInBat chGetRowIt em (' notExistTa ble ',
rows_to_ge t , columns_to _get , cond , 1 ))

try :
    result = client . batch_get_ row ( request )
    print (' Result status : % s ' %( result . is_all_suc ceed
    ()))

    table_resu lt_0 = result . get_result _by_table ( table_name )
    table_resu lt_1 = result . get_result _by_table ('
notExistTa ble ')
    print (' Check first table \' s result :')
    for item in table_resu lt_0 :
        if item . is_ok :
            print (' Read succeed , PrimaryKey : % s ,
Attributes : % s ' %( item . row . primary_ke y , item . row .
attribute_ columns ))
        else :
            print (' Read failed , error code : % s , error
message : % s ' %( item . error_code , item . error_mess age ))
            print (' Check second table \' s result :')
            for item in table_resu lt_1 :
                if item . is_ok :
                    print (' Read succeed , PrimaryKey : % s ,
Attributes : % s ' %( item . row . primary_ke y , item . row .
attribute_ columns ))
                else :
                    print (' Read failed , error code : % s , error
message : % s ' %( item . error_code , item . error_mess age ))
            # The client is abnormal , which is typically due to
            incorrect parameters or a network error .
            except OTSClientE rror as e :
                print " get row failed , http_statu s : % d , error_mess
age : % s " %( e . get_http_s tatus (), e . get_error_ message ())
            # The server is abnormal , which is typically due to
            incorrect parameters or a network error .
            except OTSService Error as e :
                print " get row failed , http_statu s : % d , error_code :
% s , error_mess age : % s , request_id : % s " %( e . get_http_s
tatus (), e . get_error_ code (), e . get_error_ message (), e .
get_reques t_id ())

```

Obtain code details at [BatchGetRow@GitHub](#).

BatchWriteRow

Inserts, modifies, or deletes several data rows in batches in one or more tables.

The BatchWriteRow operation is essentially a set of multiple PutRow, UpdateRow, and DeleteRow operations. Each operation is executed, results are returned, and capacity units are consumed independently.

API

```

"""
    Description: This operation batch modifies data
    from multiple rows.
    request = MultiTable InBatchWriteRowItem()
    request.add(TableInBatchWriteRowItem(table0,
    row_items))
    request.add(TableInBatchWriteRowItem(table1,
    row_items))
    response = client.batch_write_row(request)
    ``response`` indicates the returned results of
    the tablestore.metadata.BatchWriteRowResponse type.
"""
def batch_write_row(self, request):

```

Examples

In this example, data is batch written.

```

put_row_items = []
## Add rows by PutRow.
for i in range(0, 10):
    primary_key = [('gid', i), ('uid', i + 1)]
    attribute_columns = [('name', 'somebody' + str(i)),
    ('address', 'somewhere' + str(i)), ('age', i)]
    row = Row(primary_key, attribute_columns)
    condition = Condition(RowExistenceExpectation.IGNORE)
    item = PutRowItem(row, condition)
    put_row_items.append(item)

## Add rows by UpdateRow.
for i in range(10, 20):
    primary_key = [('gid', i), ('uid', i + 1)]
    attribute_columns = {'put': [('name', 'somebody' + str(i)),
    ('address', 'somewhere' + str(i)), ('age', i)]}
    row = Row(primary_key, attribute_columns)
    condition = Condition(RowExistenceExpectation.IGNORE, SingleColumnCondition("age", i, ComparatorType.EQUAL))
    item = UpdateRowItem(row, condition)
    put_row_items.append(item)

## Add rows by DeleteRow.
delete_row_items = []
for i in range(10, 20):
    primary_key = [('gid', i), ('uid', i + 1)]
    row = Row(primary_key)
    condition = Condition(RowExistenceExpectation.IGNORE)

```

```

        item = DeleteRowItem(row, condition)
        delete_row_items.append(item)

# Construct a batch write request.
request = BatchWriteRowRequest()
request.add(TableInBatchWriteRowItem(table_name,
put_row_items))
request.add(TableInBatchWriteRowItem('notExistTable',
delete_row_items))

# Call the batch_write_row method to perform a batch
write operation. If request parameters are incorrect,
an exception is thrown. If data fails to be written
to certain rows, then internal items fail, but no
exception is thrown.
try:
    result = client.batch_write_row(request)
    print('Result status: %s'%(result.is_all_succeed()))

## Check the results of PutRow.
print('check first table\'s put results:')
succ, fail = result.get_put()
for item in succ:
    print('Put succeed, consume %s write cu.' %
item.consumed.write)
for item in fail:
    print('Put failed, error code: %s, error
message: %s'%(item.error_code, item.error_message))

## Check the results of UpdateRow.
print('check first table\'s update results:')
succ, fail = result.get_update()
for item in succ:
    print('Update succeed, consume %s write cu.'
% item.consumed.write)
for item in fail:
    print('Update failed, error code: %s, error
message: %s'%(item.error_code, item.error_message))

## Check the results of DeleteRow.
print('check second table\'s delete results:')
succ, fail = result.get_delete()
for item in succ:
    print('Delete succeed, consume %s write cu.'
% item.consumed.write)
for item in fail:
    print('Delete failed, error code: %s, error
message: %s'%(item.error_code, item.error_message))
# The client is abnormal, which is typically due to
incorrect parameters or a network error.
except OTSClientError as e:
    print("get row failed, http_status: %d, error_message: %s" % (e.get_http_status(), e.get_error_message()))
# The server is abnormal, which is typically due to
incorrect parameters or a network error.
except OTSServiceError as e:
    print("get row failed, http_status: %d, error_code: %s, error_message: %s, request_id: %s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id()))

```

Obtain code details at [BatchWriteRow@GitHub](#).

GetRange

Reads data within the specified primary key range.

API

```

"""
    Descriptio n : This operation reads data from
multiple rows in the specified range .
    `` table_name `` is the name of the table .
    `` direction `` indicates the direction of the
range ; string format , values : ' FORWARD ' or ' BACKWARD ' .
    `` inclusive_ start_prim ary_key `` is the primary
key to start from ( inclusive ) .
    `` exclusive_ end_prim ar y_key `` is the primary key
to end on ( exclusive ) .
    `` columns_to _get `` is an optional parameter that
indicates the names of the columns to read in
list format . If not entered , all columns are read .
    `` limit `` is an optional parameter that indicates
the maximum number of rows to read . If not
entered , the number of rows to read is not limited
.
    `` column_fil ter `` is an optional parameter ,
which indicates reading the rows that meet specified
conditions .
    `` max_versio n `` is an optional parameter , which
indicates the maximum number of returned versions .
max_versio n and time_range are mutually exclusive .
    `` time_range `` is an optional parameter , which
indicates the range of returned versions . time_range
and max_versio n are mutually exclusive .
    `` start_colu mn `` is an optional parameter , which
indicates the start column of the current read
operation in the case of wide row reading .
    `` end_column `` is an optional parameter , which
indicates the end column of the current read
operation in the case of wide row reading .
    `` token `` is an optional parameter , which
indicates the location of the start column of the
current read operation in the case of wide row
reading . The content is encoded in binary format and
comes from the returned results of the previous
request .

    Return : The results list for data that meet
the conditions .
    `` consumed `` indicates the CapacityUn its consumed
by the operation . It is an instance of the
tablestore . metadata . CapacityUn it class .
    `` next_start _primary_k ey `` indicates the primary
key column value from which to start the next
get_range operation ; type : dict .
    `` row_list `` indicates the list of returned row
data , in the format of [ Row , ... ] .
"""
def get_range ( self , table_name , direction ,
                inclusive_ start_prim ary_key ,
                exclusive_ end_prim ar y_key ,
                columns_to _get = None ,
                limit = None ,
                column_fil ter = None ,

```

```

max_version = None ,
time_range = None ,
start_column = None ,
end_column = None ,
token = None ):

```

Examples

Read data in the specified range.

```

# Start primary key in the query range
inclusive_start_primary_key = [('uid ', INF_MIN ), ('gid ',
INF_MIN )]

# End primary key in the query range
exclusive_end_primary_key = [('uid ', INF_MAX ), ('gid ',
INF_MAX )]

# Query all columns
columns_to_get = []

# A maximum of 90 results are returned at a time
. If 100 results are returned in total and the
limit is set to 90 during initial query , then a
maximum of 90 results and a minimum of 0 results
may be returned for the first time , but next_start
_primary_key is not None .
    limit = 90

# Set the filter
cond = CompositeCondition( LogicalOperator . AND )
cond . add_sub_condition ( SingleColumnCondition ( " address
", ' China ', Comparator Type . EQUAL ))
cond . add_sub_condition ( SingleColumnCondition ( " age ",
50 , Comparator Type . LESS_THAN ))

try :
# Call the get_range interface
consumed , next_start_primary_key , row_list , next_token
= client . get_range (
    table_name , Direction . FORWARD ,
    inclusive_start_primary_key , exclusive_
end_primary_key ,
    columns_to_get ,
    limit ,
    column_filter = cond ,
    max_version = 1
)

    all_rows = []
    all_rows . extend ( row_list )

# If the value of next_start_primary_key is not
null , more data exists and reading continues .
    while next_start_primary_key is not None :
        inclusive_start_primary_key = next_start_primary_k
ey
        consumed , next_start_primary_key , row_list ,
next_token = client . get_range (
            table_name , Direction . FORWARD ,
            inclusive_start_primary_key , exclusive_
end_primary_key ,

```

```

        columns_to_get , limit ,
        column_filter = cond ,
        max_version = 1
    )
    all_rows . extend ( row_list )

    # Print the primary key and attribute columns
    for row in all_rows :
        print ( row . primary_key , row . attribute_columns )
    print ( ' Total rows : ' , len ( all_rows ) )
    # The client is abnormal , which is typically due to
    incorrect parameters or a network error .
    except OTSClientError as e :
        print " get row failed , http_status : % d , error_message : % s " % ( e . get_http_status () , e . get_error_message () )
    # The server is abnormal , which is typically due to
    incorrect parameters or a network error .
    except OTSServiceError as e :
        print " get row failed , http_status : % d , error_code : % s , error_message : % s , request_id : % s " % ( e . get_http_status () , e . get_error_code () , e . get_error_message () , e . get_request_id () )

```

Obtain code details at: [GetRange@GitHub](#).

2.6 Error handling

Method

Currently, the Table Store Python SDK adopts the `Exception` method to handle errors. The operations is successful if the called interface does not throw an exception. If an exception is thrown, then the operation fails.



Note:

Batch operation interfaces such as `batch_get_row` and `batch_write_row` are successfully called only when the system checks that the status of each row is successful.

Exception

The Table Store python SDK has two types of exceptions: `OTSClientError` and `OTSServiceError` inherited from `Exception`.

- **OTSClientError:** Indicates an internal SDK exception, for example, incorrect parameter values or failure to return parsed results.

- **OTSServiceError:** Indicates a server error generated by parsing a server error message. OTSServiceError has the following components:
 - **get_http_status:** Returned HTTP code, for example, 200 or 404.
 - **get_error_code:** Error type string returned by Table Store.
 - **get_error_message:** Error message string returned by Table Store.
 - **get_request_id:** UUID that uniquely identifies a request. If the problem persists, save the RequestId and [open a ticket](#).

Retry

- The system retries the operation when an error occurs in the SDK. By default, the operation is retried 20 times at a maximum interval of three seconds.. For more information about how the system retries the operation on throttling errors and read-related internal server errors, see `tablestore / lib / retry . js`
- You can inherit RetryPolicy to customize a retry policy, and pass it in as a parameter when constructing OTSClient object.

SDK has the following retry policies:

- **DefaultRetryPolicy:** The default retry policy. Only the read operation is retried 20 times at a maximum interval of three seconds.
- **NoRetryPolicy:** No retry.
- **NoDelayRetryPolicy:** A retry policy without any delays. This policy must be used with caution.
- **WriteRetryPolicy:** The write operation is retried based on the default retry policy.

3 Java SDK

3.1 Preface

This topic describes how to install and use Java SDK Version 4.0.0 and later for Table Store.

Prerequisites

- You have [activated Table Store](#) and [created an AccessKey](#).
- Java SDK Version 4.0.0 and later support multiple data versions and TTL. You have learned about [TTL](#) and [multiple versions of incremental data](#).

Compatibility

Latest Java SDK version: 4.10.2.

- For Java SDKs V4.x.x: compatible
- For Java SDKs V2.x.x: incompatible

Versions

For more information about version iterations, see [Java SDK](#).

3.2 Installation

This topic describes how to install Table Store Java SDK.

Prerequisites

Applicable to JDK 6 and later versions.

Installation methods

- Import the SDK dependency into the maven project.

To use the Table Store Java SDK in Maven, you only need to add the corresponding dependency to the pom.xml file. This example uses OSS Java SDK 4.10.2. Add the following content to <dependencies>:

```
< dependency >
  < groupId > com . aliyun . openservic es </ groupId >
  < artifactId > presto - jdbc </ artifactId >
  < version > 4 . 10 . 2 </ version >
```

```
</ dependency >
```

- Install Java SDK by importing the JAR package to Eclipse

Take the 4.10.2 version as an example. Follow the steps below:

1. [Download the Java SDK package](#).
2. Decompress the Java SDK package.
3. Copy the `tablestore -< versionId >. jar` file in the decompressed folder and all the files in the lib folder to your project.
4. Right-click your project name in Eclipse. Select Properties > Java Build Path > Add JARs.
5. Select all JAR files that you copied in Step 3.

Sample programs

The Table Store Java SDK provides a variety of sample programs for your reference or use. You can decompress the downloaded SDK package and view the sample programs in the examples folder.

3.3 Initialization

The OTSClient is the client for Table Store. It provides callers with a series of methods for operating tables and reading/writing data from/to a single row or multiple rows. To use the Java SDK to initiate a request to Table Store, you must initialize an OTSClient instance and modify the default configurations of the ClientConfiguration as needed.

Determine an endpoint

An endpoint is the domain of Alibaba Cloud Table Store in a region. It supports the following format.

Example	Description
<code>http://sun.cn-hangzhou.ots.aliyuncs.com</code>	Accesses the sun instance in Hangzhou over the Internet using the HTTP protocol.
<code>https://sun.cn-hangzhou.ots.aliyuncs.com</code>	Accesses the sun instance in Hangzhou over the Internet using the HTTPS protocol.

**Notice:**

Instances can also be accessed over the intranet.

To query the endpoint where your Table Store instance is located, follow these steps:

1. Log on to the [Alibaba Cloud Table Store Console](#).
2. Go to the Instance Details page and locate the Instance Access URL, which is the endpoint of the instance.

Configure an AccessKey

To access the Alibaba Cloud Table Store service, you need a valid AccessKey for signature authentication. The following types of AccessKeys is supported:

- AccessKeyID and AccessKeySecret of the primary account. The creation process is as follows:
 1. Register an [Alibaba Cloud account](#) on the Alibaba Cloud website.
 2. Log on to the [AccessKey Console](#) to apply for an AccessKey.
- AccessKeyID and AccessKeySecret of the RAM user authorized to access Table Store. The creation process is as follows:
 1. Use the primary account to [access RAM](#) and create a RAM user or use an existing RAM user.
 2. Use the primary account to authorize the RAM user to access Table Store.

After authorization, the AccessKeyID and AccessKeySecret of the RAM user can be used to access Table Store.
- STS token for temporary access. The token acquisition process is as follows:
 1. The application server accesses the RAM/STS server to obtain a temporary AccessKeyID, AccessKeySecret, and token, and sends them to you.
 2. Use the temporary AccessKeyID, AccessKeySecret, and token to access Table Store.

Initialization

After you obtain the AccessKeyID and AccessKeySecret, you must initialize an OTSClient instance.

Create a client

When using Table Store SDKs, you must construct a client and then call the client API to access the Table Store service. The client API functions are similar to the RESTful API provided by Table Store.

The latest versions of Table Store SDKs provide two types of clients: `SyncClient` and `AsyncClient`, which are designed for synchronous APIs and asynchronous APIs respectively.

Once a synchronous API call is completed, the request has been executed. You can call synchronous APIs to learn about the various functions of Table Store. Compared to synchronous APIs, asynchronous APIs provide greater flexibility. If you have high performance requirements, you can choose between the use of asynchronous APIs and multithreading, depending on your business needs.



Note:

Both `SyncClient` and `AsyncClient` are secure threads, which contain management threads and connection resources. We do not recommend creating too many clients. Generally, one global client is sufficient.

Sample code

1. Use the default configuration to create a `SyncClient`.

```
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";

SyncClient client = new SyncClient ( endPoint ,
accessKeyId , accessKeySecret , instanceName );
```

2. Use custom configuration to create a `SyncClient`.

```
// ClientConfiguration provides multiple configuration
// items. Commonly used items are listed as follows.
ClientConfiguration clientConfiguration = new
ClientConfiguration ();
// Set the connection establishment time - out .
clientConfiguration.setConnect ionTimeout InMillise
ond ( 5000 );
// Set the socket time - out .
clientConfiguration.setSocketT imeoutInMi llisecond (
5000 );
// Set the retry policy . If this is not set
, the default retry policy is used .
clientConfiguration.setRetrySt rategy ( new
AlwaysRetr yStrategy ());
SyncClient client = new SyncClient ( endPoint ,
accessId , accessKey ,
```

```
instanceName, clientConfiguration );
```

HTTPS

Upgrade to Java 7 for HTTPS.

Multithreading

- Multithreading is supported.
- We recommend that multiple threads use the same OTSClient object.

3.4 Table-level operations

3.4.1 Overview

The Java SDK of Table Store provides multiple table-level operations as follows:

CreateTable: create a table in Table Store.

UpdateTable: modify the configuration of a specified table.

DescribeTable: check the configuration of a specified table.

ListTable: retrieve a list of the names of tables under a specified instance.

DeleteTable: delete a specified table.

3.4.2 CreateTable

You can call this operation to create a table. Specify TableMeta and TableOptions when creating the table. You can also specify ReservedThroughput as needed.



Note:

After you create the table, Table Store loads splits of the table to a specified node. The table is available for reading and writing within a few seconds. An exception may occur if you try to read from or write to the table immediately after creating the table.

Parameters

- TableMeta

TableMeta includes the TableName and List parameters.

Parameter	Definition	Description
TableName	The name of the target table.	None
List	The definition of primary keys in the target table.	- Table Store supports multiple primary key columns. The primary key columns are sorted in the order that you added them to the table. For example, PRIMARY KEY (A, B, C) and PRIMARY KEY (A, C, B) are two different primary key structures. Table Store sorts rows according to their total primary key values. - The value of the first primary key column serves as the partition key. Table Store saves data with the same partition key to the same partition. We recommend that the same partition key be 10 GB or less. Otherwise, the partition cannot be split due to the excessive size. We also recommend that you evenly distribute read and write operations among different partition keys to load balance. - You do not need to define attribute columns. Data columns in each row can be different. You can specify the name of an attribute column when writing data to the column.

· TableOptions

TableOptions includes the TTL, MaxVersions, and MaxTimeDeviation parameters.

Parameter	Definition	Description
TTL	Time To Live, the life span of data.	- Unit: seconds. - If you do not want data to expire, set TTL to -1. - Table Store checks whether data has expired based on the data timestamp , current system time ,or table TTL . Data expires and is deleted if $\text{current system time} - \text{data timestamp} > \text{table TTL}$. For more information about timestamps, see Preface. - When you write data with TTL, make sure that you specify a valid timestamp. We recommend that you set MaxTimeDeviation to specify the timestamp.
MaxTimeDeviation	The maximum time deviation between the timestamp of written data and current system time .	- The system generates a timestamp for new data by default. The system checks whether data is expired based on the timestamp and TTL. You can specify the timestamp of written data. When the deviation between the timestamp you have specified and current system time is greater than the specified TTL, the written data expires immediately. To avoid this problem, you can set MaxTimeDeviation. - Unit: seconds. - You can specify this parameter when creating a table or by using the UpdateTable operation.
MaxVersions	The maximum number of versions that each attribute column can contain.	If the number of versions of an attribute column is more than the value of MaxVersions, Table Store only keeps the latest version that you have specified for MaxVersions.

• ReservedThroughput

The configuration of reserved read and write throughput of a table.

- Table Store calculates billing according to the reserved read and write throughput you have specified in the ReservedThroughput field.
- If ReservedThroughput is higher than 0, Table Store calculates billing based on the reserved throughput and duration. The actual read and write throughput more than the reserved quota follows the Pay-As-You-Go billing method. For more information, see [Billing items and pricing](#) to avoid unexpected fees.
- The default value 0 indicates that the Pay-As-You-Go billing method is fully applicable to Table Store.
- You can set the reserved read and write throughput of capacity instances only to 0. These instances do not allow reserved throughput.

Example

```
private static void createTable ( SyncClient client ) {
    TableMeta tableMeta = new TableMeta ( TABLE_NAME );
    tableMeta . addPrimary KeyColumn ( new PrimaryKey Schema
( PRIMARY_KEY_NAME , PrimaryKey Type . STRING )); // Add a
primary key column to the primary table .
    int timeToLive = - 1 ; // The time when data is
expired . Unit : seconds . A value of - 1 indicates that
the data is never expired . For example , to keep
the data valid for one year , set the parameter to
365 * 24 * 3600 .
    int maxVersions = 3 ; // The maximum number of
versions for each column . A value of 3 indicates
that each column contains a maximum of three of the
latest versions .
    TableOptions tableOptions = new TableOptions (
timeToLive , maxVersions );
    CreateTableRequest request = new CreateTable
Request ( tableMeta , tableOptions );
    request . setReservedThroughput ( new ReservedTh
roughput ( new CapacityUnit ( 0 , 0 ))); // Set the
reserved read and write throughput . Set the parameter
to 0 only for capacity instances , and to non - zero
values for high - performance instances .
    client . createTable ( request );
}
```

```
}
```

3.4.3 UpdateTable

You can call this operation to modify TableOptions and ReservedThroughput.

Parameters

- TableOptions

TableOptions includes the TTL, MaxVersions, and MaxTimeDeviation parameters for a table.

Parameter	Definition	Description
TTL	Time To Live, the life span of data.	- Unit: seconds. - If you do not want data to expire, set TTL to -1. - Table Store checks whether data has expired based on the <code>data timestamp</code> , <code>current time</code> , and <code>table TTL</code> . Data is expired and deleted if <code>current time - data timestamp > table TTL</code> . For more information about timestamps, see Preface. - When you write data with TTL, make sure that you specify a valid timestamp. We recommend that you set MaxTimeDeviation to specify the timestamp.
MaxTimeDeviation	The maximum time deviation between the timestamp of written data and current system time .	- The system generates a timestamp for new data by default. The system checks whether data is expired based on the timestamp and TTL. You can specify the timestamp of written data. When the deviation between the timestamp you have specified and current time is greater than the specified TTL, the written data expires immediately. To avoid this problem, you can set MaxTimeDeviation. - Unit: seconds.
MaxVersions	The maximum number of versions that each attribute column can contain.	If the number of versions of an attribute column is more than the value of MaxVersions, Table Store only keeps the latest version that you have specified for MaxVersions.

- **ReservedThroughput**

The configuration of reserved read and write throughput of a table.

- The interval of changing the value of ReservedThroughput is one minute.
- Table Store calculates billing according to the reserved read and write throughput you have specified in the ReservedThroughput field.
- If ReservedThroughput is higher than 0, Table Store calculates billing based on the reserved throughput and duration. The actual read and write throughput more than the reserved quota follows the Pay-As-You-Go billing method. For more information, see [Billing items and pricing](#) to avoid unexpected fees.
- The default value 0 indicates that the Pay-As-You-Go billing method is fully applicable to Table Store.
- You can set the reserved read and write throughput of capacity instances only to 0. These instances do not allow reserved throughput.

Example

Update TTL and Max Versions for the specified table.

```
private static void updateTable ( SyncClient client ) {
    int timeToLive = -1 ;
    int maxVersions = 5 ; // Update Max Versions to 5
    TableOptions tableOptions = new TableOptions (
        timeToLive, maxVersions );
    UpdateTableRequest request = new UpdateTableRequest (
        TABLE_NAME );
    request.setTableOptionsForUpdate ( tableOptions );
    client.updateTable ( request );
}
```

3.4.4 ListTable

You can call this operation to retrieve the list of names of all tables that you have created under the current instance.

Example

```
private static void listTable ( SyncClient client ) {
    ListTableResponse response = client.listTable ();
    System.out.println (" List of tables :");
    for ( String tableName : response.getTableNames () ) {
        System.out.println ( tableName );
    }
}
```

```
}
```

3.4.5 ComputeSplitsBySize

You can call this operation to specify the concurrency level of the compute engine. The ComputeSplitsBySize operation logically divides the data in a specified table into several splits approximately in the size specified, and returns the split points among these splits and the instances where the splits are located.

Example

```
private static void describeTable ( SyncClient client ) {
    // Split data by 200 MB .
    ComputeSplitsBySizeRequest request = new ComputeSplitsBySizeRequest ( TABLE_NAME , 2 );
    ComputeSplitsBySizeResponse response = client .
computeSplitsBySize ( computeSplitsBySizeRequest );
    System . out . println ( " ConsumedCapacity = " + response .
getConsumedCapacity () . toJson () );
    System . out . println ( " PrimaryKeySchema = " + response .
getPrimaryKeySchema () );
    System . out . println ( " RequestId = " + response . getRequestId () );
    System . out . println ( " TraceId = " + response . getTraceId () );
    List < Split > splits = response . getSplits ();
    System . out . println ( " splits . size = " + splits . size () );
    Iterator < Split > iterator = splits . iterator ();
    while ( iterator . hasNext () ) {
        Split split = iterator . next ();
        System . out . println ( " split . getLocation () = " + split .
getLocation () );
        // You can inject split . getLowerBound () and
split . getUpperBound () into RangeRowQueryCriteria , and
pass RangeRowQueryCriteria to getRange () or createRangeIterator ().
        System . out . println ( " split . getLowerBound () = " +
split . getLowerBound () . toJson () );
        System . out . println ( " split . getUpperBound () = " +
split . getUpperBound () . toJson () );
    }
}
```

3.4.6 DescribeTable

You can call this operation to obtain the information about TableMeta, TableOptions, and ReservedThroughputDetails of a specified table.

The CreateTable topic describes TableMeta and TableOptions. In addition to reserved throughput values, ReservedThroughputDetails indicates the time when you last increased or decreased these values.

Example

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

```

DescribeTableRequest request = new DescribeTableRequest
( TABLE_NAME );
DescribeTableResponse response = client.describeTable
( request );
TableMeta tableMeta = response.getTableMeta ();
System.out.println (" The name of the table : " +
tableMeta.getTableNa me ());
System.out.println (" The primary key of the table
:");
for ( PrimaryKey Schema primaryKey Schema : tableMeta .
getPrimary KeyList ()) {
    System.out.println ( primaryKey Schema );
}
TableOptions tableOptions = response.getTableOptions
();
System.out.println (" TTL of the table : " +
tableOptions.getTimeToLive ());
System.out.println (" Max Versions of the table : " +
tableOptions.getMaxVersions ());
ReservedThroughputDetails reservedThroughputDetails =
response.getReservedThroughputDetails ();
System.out.println (" Reserved read throughput of the
table : "
    + reservedThroughputDetails.getCapacityUnit ().
getReadCapacityUnit ());
System.out.println (" Reserved write throughput of
the table : "
    + reservedThroughputDetails.getCapacityUnit ().
getWriteCapacityUnit ());
}

```

3.4.7 DeleteTable

You can call this operation to delete a target table under a specified instance.

Example

```

private static void deleteTable ( SyncClient client ) {
    DeleteTableRequest request = new DeleteTableRequest (
TABLE_NAME );
    client.deleteTable ( request );
}

```

3.4.8 CreateIndex

You can call this operation to create a global secondary index table in an existing primary table.

Description

Parameter	Definition	Description
IndexName	The name of the index table.	None
List<String> primaryKey	The type of the index table.	Table Store only supports IT_GLOBAL_INDEX .

Parameter	Definition	Description
List<String> definedColumns	The attribute columns in the index table.	This is a group of columns predefined in the primary table.
IndexType	The type of the index table.	Table Store only supports IT_GLOBAL_INDEX .
IndexUpdateMode	The mode of updating the specified index table.	Table Store only supports IUM_ASYNC_INDEX .

Example

```
private static void createIndex ( SyncClient client ) {
    IndexMeta indexMeta = new IndexMeta ( INDEX_NAME );
    // The name of the index table that you want to
    // create .
    indexMeta . addPrimary KeyColumn ( DEFINED_COLUMN_NAME_1 );
    // Add a primary key column to the index table .
    indexMeta . addPrimary KeyColumn ( PRIMARY_KEY_NAME_2 );
    // Add a primary key column to the index table .
    indexMeta . addDefined Column ( DEFINED_COLUMN_NAME_2 );
    // Add an attribute column to the index table .
    CreateIndexRequest request = new CreateIndexRequest
    ( TABLE_NAME , indexMeta , false );
    client . createIndex ( request );
}
}
```

3.4.9 DeleteIndex

You can call this operation to delete a global secondary index table that you specify in a primary table. This operation does not affect other indexes.

Example

```
private static void deleteIndex ( SyncClient client ) {
    DeleteIndexRequest request = new DeleteIndexRequest
    ( TABLE_NAME , INDEX_NAME );
    // The name of the index
    // table that you want to delete and the name of the
    // corresponding primary table .
    client . deleteIndex ( request );
}
```

```
}
```

3.4.10 CreateSearchIndex

You can call this operation to create one or more Searchindex structures in an existing primary table. You can specify the index name and index structure when creating the Searchindex structure.

Parameters

- **TableName:** the name of the table where you want to create a Searchindex structure.
- **IndexName:** the name of the Searchindex structure.
- **IndexSchema:** includes the IndexSetting and FieldSchemas parameters. You can use IndexSetting to set indexes, and use FieldSchemas to set all attributes of an index.

- IndexSetting

RoutingFields: you use this advanced optional feature to customize a routing . You can specify some primary key columns as routing fields. Table Store distributes data that is written to an index to different partitions according to the specified routing fields. The data with the same routing field value is distributed to the same partition.

- FieldSchemas

Parameter	Required	Description
FieldName	Yes	■ String type. ■ The name of a field that you want to index. ■ The attribute can be a primary key column or an attribute column.
FieldType	Yes	Enumeration type. FieldType supports the following types: ■ LONG (long integer) ■ DOUBLE (floating-point number) ■ BOOLEAN (Boolean type) ■ KEYWORD (string type) ■ TEXT (tokenized string type) ■ GEO_POINT (geographical location type) ■ NESTED (nested type)

Parameter	Required	Description
Index	No	You can specify this parameter to enable indexing the target attribute.
IndexOptions	No	The optional index setting.
Analyzer	No	The optional tokenizer setting.
EnableSort AndAgg	No	■ The optional Boolean value. ■ You can specify this parameter to enable sorting and statistics.
Store	No	■ The optional Boolean value. ■ You can specify this parameter to store the value of the target field to the Searchindex structure. ■ If you set the parameter to true, you can read data directly from the index without searching the primary table. This optimizes query performance.
Array	No	■ The optional Boolean value. ■ You can specify this parameter to indicate whether the column is an array. ■ A value of true indicates that the column is an array. Data written to the column must be a JSON array, such as ["a","b","c"]. A NESTED column is an array, and does not require the Array attribute.

- IndexSort: the optional parameter to indicate the presorting method for indexed data. You can sort data by primary keys or columns. You sort data by primary keys by default.

Examples

Example 1

You can create a Searchindex structure that consists of the `Col_Keyword` and `Col_Long` columns. Then, set the data type of these columns to KEYWORD and LONG, respectively.

```
private static void createSearchIndex ( SyncClient client )
{
    CreateSearchIndexRequest request = new CreateSearchIndexRequest ();
```

```

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

```

Example 2

Specify the IndexSort parameter.

```

private static void createSear chIndexWit hIndexSort (
SyncClient client ) {
    CreateSear chIndexReq uest request = new CreateSear
chIndexReq uest ();
    request . setTableNa me ( TABLE_NAME );
    request . setIndexNa me ( INDEX_NAME );
    IndexSchem a indexSchem a = new IndexSchem a ();
    indexSchem a . setFieldSc hemas ( Arrays . asList (
        new FieldSchem a ( " Col_Keywor d ", FieldType .
KEYWORD ). setIndex ( true ). setEnableS ortAndAgg ( true ),
        new FieldSchem a ( " Col_Long ", FieldType . LONG ).
setIndex ( true ). setEnableS ortAndAgg ( true ),
        new FieldSchem a ( " Col_Text ", FieldType . TEXT ).
setIndex ( true ),
        new FieldSchem a ( " Timestamp ", FieldType . LONG ).
setIndex ( true ). setEnableS ortAndAgg ( true ) ) );
    // Presort data by the Timestamp column . You must
index the Timestamp column and enable the EnableSort
AndAgg feature .
    indexSchem a . setIndexSo rt ( new Sort (
        Arrays . < Sort . Sorter > asList ( new FieldSort ( "
Timestamp ", SortOrder . ASC ) ) ) );
    request . setIndexSc hema ( indexSchem a );
    client . createSear chIndex ( request );
}

```

```
}
```

3.4.11 DescribeSearchIndex

You can call this operation to obtain the details of the Searchindex structure of a specified primary table. The details include Searchindex fields and index configurations.

Example

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

3.4.12 ListSearchIndex

You can call this operation to retrieve the list of all Searchindex structures under a specified table.

Example

```
private static List < SearchIndexInfo > listSearchIndex ( SyncClient client ) {
    ListSearchIndexRequest request = new ListSearchIndexRequest ();
    request . setTableName ( TABLE_NAME ); // Set the name of the table .
    return client . listSearchIndex ( request ). getIndexInfos (); // Return all Searchindex structures under the specified table .
}
```

3.4.13 DeleteSearchIndex

You can call this operation to delete an unnecessary Searchindex structure.

Example

```
private static void deleteSearchIndex ( SyncClient client ) {
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest ();
    request . setTableName ( TABLE_NAME ); // Set the name of the table .
}
```

```
request.setIndexName ( INDEX_NAME ); // Set the name
of the index .
client.deleteSearchIndex ( request ); // Use client to
delete the target Searchindex structure .
}
```

3.5 Tunnel Service

3.5.1 Overview

The Java SDK of Table Store encapsulates multiple methods for Tunnel Service.

These methods include management and control methods, and an automatic data consumption framework.

- For more information about management and control methods, see:
 - [CreateTunnel](#)
 - [ListTunnel](#)
 - [DescribeTunnel](#)
 - [DeleteTunnel](#)
- For more information about the data consumption framework, see:
 - [Quick start](#)
 - [Description of the data consumption framework](#)
 - [Configure the data consumption framework](#)

3.5.2 Quick start

This topic describes how to use the Java SDK to start Tunnel Service. Example code is attached to the end of this topic.

1. Initialize a Tunnel client.

```
// Set endPoint to the endpoint of the Table Store
instance , such as https://instance.cn-hangzhou.ots
.aliyuncs.com .
// Set accessKeyId and accessKeySecret to the
AccessKeyId and AccessKeySecret for accessing the
Table Store service .
// Set instanceName to the name of the target
instance .
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
```

```
TunnelClient tunnelClient = new TunnelClient ( endPoint
, accessKeyId , accessKeySecret , instanceName );
```

2. Create a tunnel.

Create a testing table or prepare an existing table before creating the tunnel. To create a testing table, you can use the `createTable` method in the `SyncClient` class or go to the Table Store console.

```
// You can create three types of tunnels : TunnelType
. BaseData , TunnelType . Stream , and TunnelType . BaseAndStr
eam .
// The following example shows how to create a
BaseAndStream tunnel . To create the other types of
tunnels , set TunnelType in CreateTunnelRequest to
the required type .
final String tableName = " testTable ";
final String tunnelName = " testTunnel ";
CreateTunnelRequest request = new CreateTunnelRequest (
tableName , tunnelName , TunnelType . BaseAndStr eam );
CreateTunnelResponse resp = tunnelClient . createTunnel (
request );
// Use tunnelId to initialize TunnelWorker . You can
call the ListTunnel or DescribeTunnel operation to
obtain tunnelId .
String tunnelId = resp . getTunnelId ();
System . out . println ( " Create Tunnel , Id : " + tunnelId );
```

3. Customize the data consumption callback to start automatic data consumption. For more information, see [Configure the data consumption framework](#).

```
// Customize the data consumption callback or call
the IChannelProcessor operation . Specify the process
and shutdown methods .
private static class SimpleProcessor implements
IChannelProcessor {
    @ Override
    public void process ( ProcessRecordsInput input ) {
        // ProcessRecordsInput includes the data that
you have obtained .
        System . out . println ( " Default record processor ,
would print records count " );
        System . out . println (
            String . format ( " Process % d records , NextToken
: % s " , input . getRecords (). size (), input . getNextToken
()));
        try {
            // Mock Record Process .
            Thread . sleep ( 1000 );
        } catch ( InterruptedException e ) {
            e . printStackTrace ();
        }
    }
    @ Override
    public void shutdown () {
        System . out . println ( " Mock shutdown " );
    }
}
```

```
// By default , TunnelWorkerConfig starts the thread
// pool for reading data and processing data . A single
// server starts multiple TunnelWorkers .
// We recommend that you share the same TunnelWorker
// Config . TunnelWorkerConfig provides more advanced
// parameters .

TunnelWorkerConfig config = new TunnelWorkerConfig ( new
    SimpleProcessor ());
// Configure TunnelWorker and start automatic data
// processing .
TunnelWorker worker = new TunnelWorker ( tunnelId ,
    tunnelClient , config );
try {
    worker . connectAnd Working ();
} catch ( Exception e ) {
    e . printStack Trace ();
    worker . shutdown ();
    tunnelClient . shutdown ();
}
```

4. Cautions

- By default, TunnelWorkerConfig starts the thread pool for reading data and processing data. A single server starts multiple TunnelWorkers. We recommend that you share the same TunnelWorkerConfig.
- When you create a tunnel for consuming full and incremental data, the tunnel retains incremental logs for a maximum of seven days. The specific expiration time of incremental logs is the same as that of incremental data for a table. If the tunnel does not complete the consumption of full data within seven days, the OTSTunnelExpired error occurs when the tunnel starts to consume incremental data. As a result, the tunnel cannot consume incremental data. If your estimate that the tunnel cannot complete the consumption of full data within seven days, submit a ticket or join DingTalk group 11789671 to request technical support.
- TunnelWorker requires warm-up time during initialization. The heartbeatIntervalInSec parameter in TunnelWorkerConfig determines the warm-up time. You can use the setHeartbeatIntervalInSec method in TunnelWorkerConfig to set this parameter. The default value is 30s and the minimum value is 5s. For more information, see [Description of the data consumption framework](#) and [Configure the data consumption framework](#).
- The change from BaseData channels to Stream channels in the tunnel requires initialization. The heartbeatIntervalInSec parameter determines the initialization time.
- When shutting down abnormally, for example, an exceptional exit or manual termination, TunnelWorker automatically recycles resources in the following

ways: 1. releases the thread pool. 2. automatically calls the shutdown method that you have registered for the Channel class. 3. disconnects the tunnel.

5. Appendix: example

```
import com.alicloud.openservices.tablestore.TunnelClient;
import com.alicloud.openservices.tablestore.TunnelClient;
import com.alicloud.openservices.tablestore.model.CreateTunnelRequest;
import com.alicloud.openservices.tablestore.model.CreateTunnelResponse;
import com.alicloud.openservices.tablestore.model.TunnelType;
import com.alicloud.openservices.tablestore.tunnel.worker.IChannelProcessor;
import com.alicloud.openservices.tablestore.tunnel.worker.ProcessRecordsInput;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorker;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorkerConfig;
public class TunnelQuickStart {
    private static class SimpleProcessor implements IChannelProcessor {
        @Override
        public void process ( ProcessRecordsInput input ) {
            System.out.println (" Default record processor
, would print records count ");
            System.out.println (
                String.format (" Process %d records ,
NextToken : %s ", input.getRecords().size(), input.
getNextToken()));
            try {
                // Mock Record Process .
                Thread.sleep ( 1000 );
            } catch ( InterruptedException e ) {
                e.printStackTrace ();
            }
        }
        @Override
        public void shutdown () {
            System.out.println (" Mock shutdown ");
        }
    }
    public static void main () throws Exception {
        // 1 . Initialize the Tunnel client .
        final String endPoint = "";
        final String accessKeyId = "";
        final String accessKeySecret = "";
        final String instanceName = "";
        TunnelClient tunnelClient = new TunnelClient (
            endPoint , accessKeyId , accessKeySecret , instanceName );
        // 2 . Create a tunnel . You need to use the
        createTable method in SyncClient or go to the
        Table Store console to create a testing table in
        advance .
        final String tableName = " testTable ";
        final String tunnelName = " testTunnel ";
```

```

        CreateTunnelRequest request = new CreateTunnelRequest(
            tableName, tunnelName, TunnelType.BaseAndStream);
        CreateTunnelResponse resp = tunnelClient.createTunnel(request);
        // Use tunnelId to initialize TunnelWorker. You can call the ListTunnel or DescribeTunnel operation to obtain tunnelId.
        String tunnelId = resp.getTunnelId();
        System.out.println("Create Tunnel, Id: " + tunnelId);
        // 3. Customize the data consumption callback to start automatic data consumption.
        // TunnelWorkerConfig provides more advanced parameters. For more information, see the description in the related topic.
        TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
        TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
        try {
            worker.connectAndWorking();
        } catch (Exception e) {
            e.printStackTrace();
            worker.shutdown();
            tunnelClient.shutdown();
        }
    }
}

```

3.5.3 Configure the data consumption framework

Tunnel Service uses comprehensive operations of Table Store to consume full and incremental data. You can 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.

You can use TunnelWorkerConfig to configure the Tunnel client as follows:

- The interval and timeout of heartbeats
- The interval of outputting checkpoints
- The client tag
- The custom callback for processing data
- The configuration of the thread pool for reading and processing data

Configuration details

- Heartbeats
 - HeartbeatTimeoutInSec: the interval and timeout of heartbeats. Default value : 300s. When the heartbeat times out, Tunnel Service regards that the current Tunnel client is not available, and needs to run the ConnectTunnel task again.
 - HeartbeatIntervalInSec: the interval of checking heartbeats. Default value: 30s. To detect active channels by using heartbeat requests, you can set the parameter to the minimum of 5s. However, this configuration may change the interval of updating channel status or the duration for initializing the task of automatically outputting data.
- The interval of outputting checkpoints

CheckpointIntervalInMillis: the interval of outputting checkpoints in Tunnel Service after consuming data. Unit: ms. Default value: 5,000 ms.



Note:

- Reading tasks run on different servers, various errors may occur in the process . For example, the server may restart due to environmental factors, so Tunnel Service regularly outputs checkpoints after processing data. A task continues from the last checkpoint after restart. In exceptional conditions, Tunnel Service may sequentially synchronize data for once or more times. If some data is repeated, pay attention to the service processing logic.
 - To reduce processing repeated data in the case of errors, you can increase the checkpoints. However, too many checkpoints may reduce the system throughput. Therefore, determine the checkpoint frequency based on your service features.
- ClientTag: the custom client tag that is used to generate a Tunnel client ID and differentiate Tunnel clients.
 - Custom callbacks for processing data

ChannelProcessor: the callback that you register to process data, including the process and shutdown methods.

- Configuration of resources in the thread pool for reading and processing data
 - **ReadRecordsExecutor**: the data reading resource in the thread pool. Use the default configuration if you have no special requirements.
 - **ProcessRecordsExecutor**: the data processing resource in the thread pool. Use the default configuration if you have no special requirements.

**Note:**

- When you customize the thread pool, we recommend that you make the number of threads in the pool the same as that of the channels in the tunnel. In this way, each channel can access compute resources such as CPU as soon as possible.
- In the default configuration, to guarantee the throughput, follow these rules:
 - Allocate 32 core threads in advance to guarantee the real-time throughput when processing little data or few channels:
 - Properly reduce the queue length to quickly trigger the policy for creating a thread in the pool and to timely connect more compute resources when processing a large quantity of data or channels.
 - Set the thread keep-alive time to 60s to recycle thread resources after the data size reduces.

3.5.4 CreateTunnel

You can call this operation to create one or more tunnels for a specified table. Specify the table name, tunnel name, and tunnel type when creating the tunnel.

Request parameters

- **TableName**: the name of the table that you want to create tunnels for.
- **TunnelName**: the name of the tunnel that you want to create.
- **TunnelType**: the type of the tunnel that you want to create. Valid values: **BaseData**, **Stream**, and **BaseAndStream**.

Response parameters

- **TunnelId**: the ID of the tunnel that you want to create.
- **ResponseInfo**: some other fields returned in the request.

RequestId: the GUID generated by Alibaba Cloud for the request.

Example

```
// You can create three types of tunnels : TunnelType .
// BaseData , TunnelType . Stream , and TunnelType . BaseAndStream
// The following example shows how to create a
// BaseData tunnel . To create other types of tunnels ,
// set TunnelType in CreateTunnelRequest to the required
// type .
private static void createTunnel ( TunnelClient client ,
String tunnelName ) {
    CreateTunnelRequest request = new CreateTunnelRequest (
        TableName , tunnelName , TunnelType . BaseData );
    CreateTunnelResponse resp = client . createTunnel (
        request );
    System . out . println ( " RequestId : " + resp . getRequest Id
        ( ) );
    System . out . println ( " TunnelId : " + resp . getTunnelId
        ( ) );
}
```

3.5.5 ListTunnel

You can call this operation to list the tunnels of a specified table.

Request parameters

TableName: the name of the table that you want to list tunnels for.

Response parameters

- **List<TunnelInfo>:** the list of required tunnels.
 - **TunnelId:** the ID of a required tunnel.
 - **TunnelType:** the type of the required tunnel. Valid values: BaseData, Stream, and BaseAndStream.
 - **TableName:** the name of the table where the required tunnel is located.
 - **InstanceName:** the name of the instance where the required tunnel is located.
 - **Stage:** the stage where the required tunnel is located. Valid values: InitBaseDataAndStreamShard, ProcessBaseData, and ProcessStream.
 - **Expired:** indicates whether data is expired. If a value of true is returned, request technical support.
- **ResponseInfo:** some other fields returned in the request.

RequestId: the GUID generated by Alibaba Cloud for the request.

Example

```
private static void listTunnel ( TunnelClient client ,
String tableName ) {
```

```

        ListTunnel Request request = new ListTunnel Request (
        tableName );
        \ ListTunnel Response resp = client . listTunnel ( request );
        System . out . println ( " RequestId : " + resp . getRequest Id
        ());
        for ( TunnelInfo info : resp . getTunnelI nfos ()) {
            System . out . println ( " TunnelInfo :::::" );
            System . out . println ( "\ tTunnelNam e : " + info .
            getTunnelN ame ());
            System . out . println ( "\ tTunnelId : " + info .
            getTunnelI d ());
            // The type of the required tunnel . Valid
            values : BaseData , Stream , and BaseAndStr eam .
            System . out . println ( "\ tTunnelTyp e : " + info .
            getTunnelT ype ());
            System . out . println ( "\ tTableName : " + info .
            getTableNa me ());
            System . out . println ( "\ tInstanceN ame : " + info .
            getInstanc eName ());
            // The stage where the required tunnel is
            located . Valid values : InitBaseDa taAndStrea mShard ,
            ProcessBas eData , and ProcessStr eam .
            System . out . println ( "\ tStage : " + info . getStage ());
            // Indicates whether data is expired . If a
            value of true is returned , request technical support .
            System . out . println ( "\ tExpired : " + info . isExpired
            ());
        }
    }
}

```

3.5.6 DescribeTunnel

You can call this operation to obtain the information of channels in a specified tunnel. Currently, a channel corresponds to a split for the TableStore Stream operation.

Request parameters

- **TableName:** the name of the table for the specified tunnel.
- **TunnelName:** the name of the specified tunnel.

Response parameters

- **TunnelConsumePoint:** the latest time when the tunnel consumes incremental data . The time equals that when the last channel in the tunnel consumes data. Default value: January 1, 1970 (UTC).

- **TunnelInfo**: the information of the specified tunnel.
 - **TunnelId**: the ID of the specified tunnel.
 - **TunnelType**: the type of the specified tunnel. Valid values: **BaseData**, **Stream**, and **BaseAndStream**.
 - **TableName**: the name of the table where the specified tunnel is located.
 - **InstanceName**: the name of the instance where the specified tunnel is located.
 - **Stage**: the stage where the specified tunnel is located. Valid values: **InitBaseDataAndStreamShard**, **ProcessBaseData**, and **ProcessStream**.
 - **Expired**: indicates whether data is expired. If a value of **true** is returned, request technical support.
- **List<ChannelInfo>**: the list of channels in the specified tunnel.
 - **ChannelId**: the ID of a channel.
 - **ChannelType**: the type of a channel. Valid values: **BaseData** and **Stream**.
 - **ChannelStatus**: the status of a channel. Valid values: **WAIT**, **OPEN**, **CLOSING**, **CLOSE**, and **TERMINATED**.
 - **ClientId**: the ID of the Tunnel client. By default, **ClientId** includes a client host name that is customized in **TunnelWorkerConfig** and that is joined with a random string.
 - **ChannelConsumePoint**: the latest time when a channel consumes incremental data. Default value: January 1, 1970 (UTC). This parameter is not used for full data consumption.
 - **ChannelCount**: the number of data items that a channel synchronizes.
- **ResponseInfo**: some other fields returned in the request.

RequestId: the GUID generated by Alibaba Cloud for the request.

Example

```
// The ConsumePoint and Recovery Point Objective (RPO)
// attributes are used for incremental data consumption,
// rather than for full data consumption.
// Stream tunnel: The Stage parameter in TunnelInfo is
// set to ProcessStream. Stream channel: The ChannelType
// parameter in ChannelInfo is set to Stream.
private static void describeTunnel(TunnelClient client,
    String tableName, String tunnelName) {
    DescribeTunnelRequest request = new DescribeTunnelRequest(
        tableName, tunnelName);
    DescribeTunnelResponse resp = client.describeTunnel(request);
    System.out.println("RequestId: " + resp.getRequestId());
}
```

```

// The latest time when the tunnel consumes
incremental data. The time equals that when the
last channel in the tunnel consumes data. Default
value: January 1, 1970 (UTC).
System.out.println(" TunnelConsumePoint: " + resp.
getTunnelConsumePoint());
System.out.println(" TunnelInfo: " + resp. getTunnelI
nfo());
for (ChannelInfo ci : resp.getChannelInfos()) {
    System.out.println(" ChannelInfo :::::");
    System.out.println("\tChannelId: " + ci.getChannel
Id());
    // The type of a channel. Valid values:
    BaseData and Stream.
    System.out.println("\tChannelType: " + ci.
getChannelType());
    // The ID of the Tunnel client. By default,
    the parameter includes a client host name joined with
    a random string.
    System.out.println("\tClientId: " + ci.getClientI
d());
    // The latest time when the channel consumes
    incremental data.
    System.out.println("\tChannelConsumePoint: " + ci.
getChannelConsumePoint());
    // The number of data items that the channel
    synchronizes.
    System.out.println("\tChannelCount: " + ci.
getChannelCount());
}
}

```

3.5.7 DeleteTunnel

You can call this operation to delete a tunnel for a table by specifying the required table name and tunnel name.

Request parameters

- **TableName(required):** the name of the table where you delete the specified tunnel.
- **TunnelName(required):** the name of the tunnel that you want to delete.

Response parameters

ResponseInfo: some other fields returned in the request.

RequestId: the GUID generated by Alibaba Cloud for the request.

Example

```

private static void deleteTunnel(TunnelClient client,
String tableName, String tunnelName) {
    DeleteTunnelRequest request = new DeleteTunnelRequest(
tableName, tunnelName);
    DeleteTunnelResponse resp = client.deleteTunnel(
request);
    System.out.println(" RequestId: " + resp.getRequestId
());
}

```

```
}
```

3.6 Search Index-based operations

3.6.1 Query by exact match

TermQuery

You can use TermQuery to query data that exactly matches the specified value of a field. When a table contains a TEXT string, Table Store tokenizes the string and exactly matches any of the tokens.

For example, Table Store tokenizes Text string "tablestore is cool" into "tablestore", "is", and "cool". When you specify any of these tokens as a query string, you can retrieve the query result that contains the token.

Example

```
/**
 * Search the table for rows where the value of
 * Col_Keyword exactly matches "hangzhou".
 * @param client
 */
private static void termQuery ( SyncClient client ) {
    SearchQuery searchQuery = new SearchQuery ();
    TermQuery termQuery = new TermQuery (); // Set the
    query type to TermQuery .
    termQuery . setFieldName ( " Col_Keyword " ); // Set the
    name of the field that you want to match .
    termQuery . setTerm ( ColumnValue . fromString ( " hangzhou
    " )); // Set the value that you want to match .
    searchQuery . setQuery ( termQuery );
    SearchRequest searchRequest = new SearchRequest (
    TABLE_NAME , INDEX_NAME , searchQuery );

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

    SearchResponse resp = client . search ( searchRequest );
    System . out . println ( " Row : " + resp . getRows () );
    // You can check whether NextToken is null . If
    not , you can use NextToken to continue reading data .
```

```
}
```

TermsQuery

You can use `TermsQuery` to query data that exactly matches the specified field values. `TermQuery` is similar to `TermsQuery`. The difference is that `TermsQuery` supports multiple terms. You can retrieve query results that match any of these terms.

```
/**
 * Search the table for rows where the value of
 * Col_Keyword exactly matches "hangzhou" or "shanghai".
 * TermsQuery supports multiple terms. You can retrieve
 * query results that match any of these terms.
 * @param client
 */
private static void termsQuery ( SyncClient client ) {
    SearchQuery searchQuery = new SearchQuery ();
    TermsQuery termsQuery = new TermsQuery (); // Set the
    query type to TermQuery.
    termsQuery . setFieldName ( " Col_Keyword " ); // Set the
    name of the field that you want to match .
    termsQuery . setTerms ( Arrays . asList ( ColumnValue .
    fromString ( " hangzhou " ),
        ColumnValue . fromString ( " shanghai " ) ) ); // Set
    the value that you want to match .
    searchQuery . setQuery ( termsQuery );
    SearchRequest searchRequest = new SearchRequest (
    TABLE_NAME , INDEX_NAME , searchQuery );

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

    SearchResponse resp = client . search ( searchRequest );
    System . out . println ( " Row : " + resp . getRows () );
    // You can check whether NextToken is null . If
    not , you can use NextToken to continue reading data .
}
```

3.6.2 Query by match

MatchAllQuery

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

```
/**
 * Use MatchAllQuery to query the total number of
 * rows in a table .
 * @param client
 */
private static void matchAllQuery ( SyncClient client ) {
    SearchQuery searchQuery = new SearchQuery ();
    searchQuery . setQuery ( new MatchAllQuery () ); // Set
    the query type to MatchAllQuery .
}
```

```

    * In the MatchAllQuery - based query result , the
    value of TotalCount is the total number of rows in
    a table .
    * To return only the total number of rows without
    any specific data , you can set Limit to 0 . Then
    , Table Store returns no data in the rows .
    */
    searchQuery . setLimit ( 0 );
    searchQuery . setGetTotalCount ( true );
    SearchRequest searchRequest = new SearchRequest (
    TABLE_NAME , INDEX_NAME , searchQuery );
    SearchResponse resp = client . search ( searchRequest );
    /**
    * Check whether Table Store returns matched data
    from all partitions . When the value of isAllSuccess
    is false , Table Store may fail to query some
    partitions and return a part of data .
    */
    if ( ! resp . isAllSuccess () ) {
        System . out . println ( " NotAllSuccess !" ) ;
    }
    System . out . println ( " IsAllSuccess : " + resp .
    isAllSuccess () );
    System . out . println ( " TotalCount : " + resp . getTotalCo
    unt () ); // The total number of rows .
    System . out . println ( resp . getRequest Id () );
}

```

MatchQuery

You can use MatchQuery to query a table based on approximate matches. For example, you can search "this is". Table Store returns query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".

```

/**
* Search the table for rows where the value of
Col_Keyword matches "hangzhou". Table Store returns
matched rows and the total number of matched rows .
* @param client
*/
private static void matchQuery ( SyncClient client ) {
    SearchQuery searchQuery = new SearchQuery ();
    MatchQuery matchQuery = new MatchQuery (); // Set the
query type to MatchQuery .
    matchQuery . setFieldName ( " Col_Keyword " ); // Set the
name of the field that you want to match .
    matchQuery . setText ( " hangzhou " ); // Set the value that
you want to match .
    searchQuery . setQuery ( matchQuery );
    searchQuery . setOffset ( 0 ); // Set Offset to 0 .
    searchQuery . setLimit ( 20 ); // Set Limit to 20 to
return a maximum of 20 rows .
    searchQuery . setGetTotalCount ( true );
    searchQuery . setSort ( new Sort ( Arrays . asList ( new
ScoreSort () ) ) ); // Sort the result by match score .
    SearchRequest searchRequest = new SearchRequest (
    TABLE_NAME , INDEX_NAME , searchQuery );
    SearchResponse resp = client . search ( searchRequest );
    System . out . println ( " TotalCount : " + resp . getTotalCo
unt () );
}

```

```

        System . out . println ( " Row : " + resp . getRows ()); // If
        you do not set columnsToGet , Table Store only
        returns primary keys by default .

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

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

```

MatchPhraseQuery

You can use MatchPhraseQuery to search matched phrases. MatchPhraseQuery is similar to MatchQuery. The difference is that MatchPhraseQuery matches a phrase as a whole. For example, if you search the phrase "this is", you can obtain query results such as "..., this is tablestore" and "this is a table". You cannot obtain query results such as "this table is ..." and "is this a table".

```

/**
 * Search the table for rows where the value of
 * Col_Text matches "hangzhou shanghai". Table Store returns
 * the total number of rows that match the phrase as
 * a whole and matched rows in this query .
 * @param client
 */
private static void matchPhraseQuery ( SyncClient client )
{
    SearchQuery searchQuery = new SearchQuery ();
    MatchPhraseQuery matchPhraseQuery = new MatchPhraseQuery (); // Set the query type to MatchPhraseQuery .
    matchPhraseQuery . setFieldName ( " Col_Text " ); // Set the
    name of the field that you want to match .
    matchPhraseQuery . setText ( " hangzhou shanghai " ); // Set
    the value that you want to match .
    searchQuery . setQuery ( matchPhraseQuery );
    searchQuery . setOffset ( 0 ); // Set Offset to 0 .
    searchQuery . setLimit ( 20 ); // Set Limit to 20 to
    return up to 20 rows .
    searchQuery . setGetTotalCount ( true );
    searchQuery . setSort ( new Sort ( Arrays . asList ( new
    ScoreSort ());)); // Sort the result by match score .
    SearchRequest searchRequest = new SearchRequest (
    TABLE_NAME , INDEX_NAME , searchQuery );
    SearchResponse resp = client . search ( searchRequest );
    System . out . println ( " TotalCount : " + resp . getTotalCount ());
    System . out . println ( " Row : " + resp . getRows ()); //
    Return primary keys only by default .

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

```

```

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

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

```

3.6.3 PrefixQuery

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

Example

```

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

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

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

```

```
}
```

3.6.4 RangeQuery

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

Example

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

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

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

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

3.6.5 WildcardQuery

You can use WildcardQuery to query data that matches wildcard characters.

You can specify a value you want to match as a string that consists of one or more wildcard characters. An asterisk (*) is interpreted as a number of characters or an empty string. A question mark (?) is interpreted as any single character. For example, when you search the string "table*e", you can retrieve query results such as "tablestore". The string specified as a filtering condition cannot start with an asterisk (*).

Example

```
/**
 * Search the table for rows where the value of
 * Col_Keyword matches "hang * u".
 * @param client
 */
private static void wildcardQuery ( SyncClient client ) {
    SearchQuery searchQuery = new SearchQuery ();
    WildcardQuery wildcardQuery = new WildcardQuery ();
    // Set the query type to WildcardQuery .
    wildcardQuery . setFieldName ( " Col_Keyword " );
    wildcardQuery . setValue ( " hang * u " ); // Specify a
    string that contains one or more wildcard characters
    in wildcardQuery .
    searchQuery . setQuery ( wildcardQuery );
    searchQuery . setGetTotalCount ( true );
    SearchRequest searchRequest = new SearchRequest (
    TABLE_NAME , INDEX_NAME , searchQuery );

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

    SearchResponse resp = client . search ( searchRequest );

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

3.6.6 Geographic location-based query

Search Index-based queries support three geographic location-based queries: GeoBoundingBoxQuery, GeoDistanceQuery, and GeoPolygonQuery.

GeoBoundingBoxQuery

You can use GeoBoundingBoxQuery to query data that falls within a geographic rectangular area.

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

```
/**
 * The data type of Col_GeoPoint is GeoPoint . You
 * can obtain the rows where the value of Col_GeoPoi
 * nt falls within the range of a geographic rectangula
 * r area . For the geographic rectangular area , the
 * upper - left vertex is " 10 , 0 " and the lower - right
 * vertex is " 0 , 10 ".
 * @param client
```

```

*/
public static void geoBoundin gBoxQuery ( SyncClient client
) {
    SearchQuer y searchQuer y = new SearchQuer y ();
    GeoBoundin gBoxQuery geoBoundin gBoxQuery = new
GeoBoundin gBoxQuery (); // Set the query type to
GeoBoundin gBoxQuery .
    geoBoundin gBoxQuery . setFieldNa me (" Col_GeoPoi nt "); //
Set the name of the field that you want to match
.
    geoBoundin gBoxQuery . setTopLeft (" 10 , 0 "); // Specify
coordinate s for the upper - left vertex of the
geographic rectangula r area .
    geoBoundin gBoxQuery . setBottomR ight (" 0 , 10 "); //
Specify coordinate s for the lower - right vertex of
the geographic rectangula r area .
    searchQuer y . setQuery ( geoBoundin gBoxQuery );
    searchQuer y . setGetTota lCount ( true );

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

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

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

```

GeoDistanceQuery

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

```

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

```

```

        searchQuery.setGetTotalCount(true);

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

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

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

```

GeoPolygonQuery

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

```

/**
 * Search the table for rows where the value of
 * Col_GeoPoint falls within a specified geographic
 * polygon area.
 * @param client
 */
public static void geoPolygonQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoPolygonQuery geoPolygonQuery = new GeoPolygonQuery(); // Set the query type to GeoPolygonQuery.
    geoPolygonQuery.setFieldName("Col_GeoPoint");
    geoPolygonQuery.setPoints(Arrays.asList("0, 0", "5, 5", "5, 0")); // Specify coordinates for vertices of
    a geographic polygon.
    searchQuery.setQuery(geoPolygonQuery);
    searchQuery.setGetTotalCount(true);

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

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

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

```

```
}
```

3.6.7 BoolQuery

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

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

You can configure the following parameters for BoolQuery:

```
/**
 * The document must exactly match all subqueries .
 */
List < Query > mustQueries ;
/**
 * The document must not match any subquery .
 */
List < Query > mustNotQueries ;
/**
 * The document must exactly match all subfilters . A
 * filter is similar to a query . But the system does
 * not calculate scores when you use the filter .
 */
List < Query > filterQueries ;
/**
 * You can set minimumShouldMatch to specify the
 * minimum number of shouldQueries subqueries that the
 * query result must match . The higher relevance results
 * in a higher score .
 */
List < Query > shouldQueries ;
/**
 * Set minimumShouldMatch . Default value : 1 .
 */
Integer minimumShouldMatch ;
```

BoolQuery and subqueries of BoolQuery can be a query of any type. Therefore, you can use BoolQuery to perform complexly combined query operations.

Example

```
/**
 * Use BoolQuery to query data that matches a
 * combination of conditions .
 * @param client
 */
public static void boolQuery ( SyncClient client ) {
    /**
```

```

    * Condition 1 : use RangeQuery to query data where
    the value of Col_Long is greater than 3 .
    */
    RangeQuery rangeQuery = new RangeQuery ();
    rangeQuery . setFieldName ( " Col_Long " );
    rangeQuery . greaterThan ( ColumnValue . fromLong ( 3 ));

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

    SearchQuery searchQuery = new SearchQuery ();
    {
        /**
        * Create a query of BoolQuery type where the
        result meets Conditions 1 and 2 at the same time .
        */
        BoolQuery boolQuery = new BoolQuery ();
        boolQuery . setMustQueries ( Arrays . asList ( rangeQuery
        , matchQuery ));
        searchQuery . setQuery ( boolQuery );
        searchQuery . setGetTotalCount ( true );
        SearchRequest searchRequest = new SearchRequest (
        TABLE_NAME , INDEX_NAME , searchQuery );
        SearchResponse resp = client . search ( searchRequest
        );
        System . out . println ( " TotalCount : " + resp .
        getTotalCount ()); // The total number of matched rows
        instead of the number of returned rows .
        System . out . println ( " Row : " + resp . getRows ());
    }

    {
        /**
        * Create a query of BoolQuery type where the
        result meets at least one of Conditions 1 and 2 .
        */
        BoolQuery boolQuery = new BoolQuery ();
        boolQuery . setShouldQueries ( Arrays . asList ( rangeQuery
        , matchQuery ));
        boolQuery . setMinimumShouldMatch ( 1 ); // Specify
        that the result meets at least one of the
        conditions .
        searchQuery . setQuery ( boolQuery );
        searchQuery . setGetTotalCount ( true );
        SearchRequest searchRequest = new SearchRequest (
        TABLE_NAME , INDEX_NAME , searchQuery );
        SearchResponse resp = client . search ( searchRequest
        );
        System . out . println ( " TotalCount : " + resp .
        getTotalCount ()); // The total number of matched rows
        instead of the number of returned rows .
        System . out . println ( " Row : " + resp . getRows ());
    }
}

```

```
}
```

3.6.8 NestedQuery

You can use `NestedQuery` to query data based on Nested fields. You cannot use Nested fields directly to query data. Subqueries must be nested in `NestedQuery`. In this query, you need to specify a subquery that can be any type of query and the path of a Nested field.

Example

```
/**
 * The NESTED column contains nested_1 and nested_2 .
 * You need to search the col_nested . nested_1 column for
 * data that matches " tablestore ".
 * @param client
 */
private static void nestedQuery ( SyncClient client ) {
    SearchQuery searchQuery = new SearchQuery ();
    NestedQuery nestedQuery = new NestedQuery (); // Set
    the query type to NestedQuery .
    nestedQuery . setPath ( " col_nested " ); // Set the path
    of the NESTED field .
    TermQuery termQuery = new TermQuery (); // Specify a
    subquery for NestedQuery .
    termQuery . setFieldName ( " col_nested . nested_1 " ); // Set
    the name of the field that you want to match . The
    field name must contain the prefix of the Nested
    column .
    termQuery . setTerm ( ColumnValue . fromString ( " tablestore
    " ) ); // Set the value that you want to match .
    nestedQuery . setQuery ( termQuery );
    nestedQuery . setScoreMode ( ScoreMode . None );
    searchQuery . setQuery ( nestedQuery );
    searchQuery . setGetTotalCount ( true );
    SearchRequest searchRequest = new SearchRequest (
    TABLE_NAME , INDEX_NAME , searchQuery );

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

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

```
}
```

3.6.9 Sorting and paging

IndexSort

Table Store sorts matched data based on the `IndexSort` field value in Search Index-based queries. Indexes of NESTED type does not support `IndexSort`. The default `IndexSort` field value is the name of a primary key column. You can define the `IndexSort` field when you create an index.

`IndexSort` determines the default order of the data that Table Store returns in Search Index-based queries. If you do not specify the `IndexSort` field, Table Store returns the query result in the order of primary key columns.

Specify a sorting method

You can use `Sorter` to specify a sorting method for each query. Search Index-based queries support the following sorting methods. You can also use multiple sorting methods and prioritize each method as needed.

ScoreSort

You can use `ScoreSort` to sort the query result by score. `ScoreSort` is applicable to relevance scenarios such as full-text indexing. You must set `ScoreSort` to sort the query result by relevance. Otherwise, Table Store returns the query result based on the `IndexSort` field.

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

PrimaryKeySort

You can use `PrimaryKeySort` to sort the query result by primary key column.

```
Forward order :
SearchQuery query = new SearchQuery();
query.setSort(new Sort(Arrays.asList(new
    PrimaryKeySort())));

Inverted order :
SearchQuery query = new SearchQuery();
query.setSort(new Sort(Arrays.asList(new
    PrimaryKeySort(SortOrder.DESC))));
```

FieldSort

You can use `FieldSort` to sort the query result by a specified column.

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

Prioritize columns to sort the query result.

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

GeoDistanceSort

You can use `GeoDistanceSort` to sort the query result by geographic location.

```
SearchQuery query = new SearchQuery();
// Sort the result by the distance from the value
// of the geo column of GEOPOINT type to the "0, 0"
// point.
Sort.Sorter sorter = new GeoDistanceSort("geo", Arrays
    .asList("0", "0"));
query.setSort(new Sort(Arrays.asList(sorter)));
```

Paging

Use LIMIT and OFFSET

When the total number of target rows is less than 2,000, to scan 2,000 pages or fewer and return the result, you can set `LIMIT` and `OFFSET` in the following way: `LIMIT+OFFSET <= 2000`.

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

Use a token

If Table Store does not complete reading all required data, Table Store returns `NextToken`. You can use `NextToken` to continue reading the subsequent data. You cannot set the sorting method if you use a token. When you use the token, the sorting method is the same as that used in the previous request. The system sorts data based on the `IndexSort` field by default or based on the method that you have specified. Also, you cannot set `Offset` if you use a token. In this case, the system scans data page by page. This results in a slow query.

```
/**
```

```

* If you use a token for paging, in this example
, the system reads all data and places the data in
a list.
* @param client
*/
private static void readMoreRowsWithToken ( SyncClient
client ) {
    SearchQuery searchQuery = new SearchQuery ();
    searchQuery.setQuery ( new MatchAllQuery () );
    searchQuery.setGetTotalCount ( true );
    SearchRequest searchRequest = new SearchRequest (
TABLE_NAME , INDEX_NAME , searchQuery );
    SearchResponse resp = client.search ( searchRequest );
    if ( ! resp.isAllSuccessful () ) {
        throw new RuntimeException ( " not all success " );
    }
    List < Row > rows = resp.getRows ();
    while ( resp.getNextToken () != null ) { // The system
stops reading when NextToken is null. This indicates
that the system reads all data.
        // Set Token in this request to the NextToken
value in the previous response.
        searchRequest.getSearchQuery().setToken ( resp.
getNextToken () );
        resp = client.search ( searchRequest );
        if ( ! resp.isAllSuccessful () ) {
            throw new RuntimeException ( " not all success
");
        }
        rows.addAll ( resp.getRows () );
    }
    System.out.println ( " RowSize : " + rows.size () );
    System.out.println ( " TotalCount : " + resp.getTotalCo
unt () );
}

```

3.7 Single-row operations

Table Store provides the following single-row operation APIs: PutRow, GetRow, UpdateRow, and DeleteRow.

PutRow

The PutRow API is used to insert a row of data. If this row already exists, it is overwritten.

When performing PutRow, you can use the Conditional Update function to set conditions for the existence of the row to be written or values in certain columns of the existing row. For more information, see [Conditional Update](#).

Example 1

In this example, a row with 10 attribute columns is written, and one version for each column is entered. The version number (timestamp) is specified by the server.

```
private static void putRow ( SyncClient client , String
pkValue ) {
    // Build primary key .
    PrimaryKey Builder primaryKey Builder = PrimaryKey Builder
    . createPrim aryKeyBuil der ();
    primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE Y_NAME
, PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey primaryKey = primaryKey Builder . build ();

    RowPutChan ge rowPutChan ge = new RowPutChan ge (
TABLE_NAME , primaryKey );

    // Add attribute columns
    for ( int i = 0 ; i < 10 ; i ++ ) {
        rowPutChan ge . addColumn ( new Column ( " Col " + i ,
ColumnValu e . fromLong ( i ));
    }

    client . putRow ( new PutRowRequ est ( rowPutChan ge ));
}
```

Example 2

In this example, a row with 10 attribute columns is written, and three versions for each column is entered. The version number (timestamp) is specified by the client.

```
private static void putRow ( SyncClient client , String
pkValue ) {
    // Build primary key .
    PrimaryKey Builder primaryKey Builder = PrimaryKey Builder
    . createPrim aryKeyBuil der ();
    primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE Y_NAME
, PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey primaryKey = primaryKey Builder . build ();

    RowPutChan ge rowPutChan ge = new RowPutChan ge (
TABLE_NAME , primaryKey );

    // Add attribute columns
    long ts = System . currentTim eMillis ();
    for ( int i = 0 ; i < 10 ; i ++ ) {
        for ( int j = 0 ; j < 3 ; j ++ ) {
            rowPutChan ge . addColumn ( new Column ( " Col " + i
, ColumnValu e . fromLong ( j ), ts + j ));
        }
    }

    client . putRow ( new PutRowRequ est ( rowPutChan ge ));
}
```

Example 3

In this example, if no row exists, the data is written.

```
private static void putRow ( SyncClient client , String
pkValue ) {
```

```

// Build primary key .
PrimaryKey Builder primaryKey Builder = PrimaryKey Builder
.createPrimaryKeyBuilder ();
primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KEY_NAME
, PrimaryKey Value . fromString ( pkValue ));
PrimaryKey primaryKey = primaryKey Builder . build ();

RowPutChange rowPutChange = new RowPutChange (
TABLE_NAME , primaryKey );

// Expect the original row does not exist
rowPutChange . setCondition ( new Condition ( RowExistenceExpectation . EXPECT_NOT_EXIST ));

// Add attribute columns
long ts = System . currentTimeMillis ();
for ( int i = 0 ; i < 10 ; i ++ ) {
    for ( int j = 0 ; j < 3 ; j ++ ) {
        rowPutChange . addColumn ( new Column ( " Col " + i
, ColumnValue . fromLong ( j ), ts + j ));
    }
}

client . putRow ( new PutRowRequest ( rowPutChange ));
}

```

Example 4

In this example, the original row is expected to exist and to write data when the Col0 value is greater than 100.

```

private static void putRow ( SyncClient client , String
pkValue ) {
    // Build primary key .
    PrimaryKey Builder primaryKey Builder = PrimaryKey Builder
.createPrimaryKeyBuilder ();
    primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KEY_NAME
, PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey primaryKey = primaryKey Builder . build ();

    RowPutChange rowPutChange = new RowPutChange (
TABLE_NAME , primaryKey );

    // Expect the original row to exist and write
data when the Col0 value is greater than 100
    Condition condition = new Condition ( RowExistenceExpectation . EXPECT_EXIST );
    condition . setColumnCondition ( new SingleColumnValueCondition ( " Col0 ",
SingleColumnValueCondition . CompareOperator .
GREATER_THAN , ColumnValue . fromLong ( 100 ));
    rowPutChange . setCondition ( condition );

    // Add attribute columns
    long ts = System . currentTimeMillis ();
    for ( int i = 0 ; i < 10 ; i ++ ) {
        for ( int j = 0 ; j < 3 ; j ++ ) {
            rowPutChange . addColumn ( new Column ( " Col " + i
, ColumnValue . fromLong ( j ), ts + j ));
        }
    }
}

```

```

        client . putRow ( new      PutRowRequ  est ( rowPutChan  ge ));
    }

```

GetRow

The single-row GetRow API is used to read a single row of data. It has the following parameters:

- **PrimaryKey:** The primary key of the row to read. It is a required parameter.
- **ColumnsToGet:** The set of columns to read. If not set, all columns are read.
- **MaxVersions:** The maximum number of versions to read. At least one of the MaxVersions and TimeRange parameters must be set.
- **TimeRange:** The range of version numbers to read. At least one of the MaxVersions and TimeRange parameters must be set.
- **Filter:** The filters applied. Filters are used by the server to filter the read results again.

Example 1

In this example, the latest version of single row is read and the specific ColumnsToGet is set.

```

private static void getRow ( SyncClient  client ,
String  pkValue ) {
    // Build primary key .
    PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
    primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey  primaryKey = primaryKey Builder . build
();

    // Read a row .
    SingleRowQ ueryCriter ia criteria = new
SingleRowQ ueryCriter ia ( TABLE_NAME , primaryKey );
    // Set the latest version to be read .
    criteria . setMaxVers ions ( 1 );
    GetRowResp onse getRowResp onse = client . getRow (
new GetRowRequ est ( criteria ));
    Row row = getRowResp onse . getRow ();

    System . out . println ( " Read complete , result :");
    System . out . println ( row );

    // Set the columns to read
    criteria . addColumns ToGet ( " Col0 " );
    getRowResp onse = client . getRow ( new GetRowRequ
est ( criteria ));
    row = getRowResp onse . getRow ();

    System . out . println ( " Read complete , result :");
    System . out . println ( row );
}

```

```
}
```

Example 2

In this example, a filter is set.

```
private static void getRow ( SyncClient client , String
pkValue ) {
    // Build primary key .
    PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
    primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey primaryKey = primaryKey Builder . build
();

    // Read a row .
    SingleRowQ ueryCriter ia criteria = new
SingleRowQ ueryCriter ia ( TABLE_NAME , primaryKey );
    // Set the latest version to be read .
    criteria . setMaxVers ions ( 1 );

    // Set the filter ; when the Col0 value is
0 , this row is returned .
    SingleColu mnValueFil ter singleColu mnValueFil ter
= new SingleColu mnValueFil ter ( " Col0 " ,
SingleColu mnValueFil ter . CompareOpe rator .
EQUAL , ColumnValu e . fromLong ( 0 ));
    // If the column Col0 does not exist , the
data will not be returned .
    singleColu mnValueFil ter . setPassIfM issing ( false
);
    criteria . setFilter ( singleColu mnValueFil ter );

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

    System . out . println ( " Read complete , result : " );
    System . out . println ( row );
}
```

UpdateRow

The UpdateRow API is used to update a single row of data. If no row exists, a new row is added.

The update operation includes write column, delete column, and delete column version operations.

When performing UpdateRow, you can use the Conditional Update function to set conditions for the existence of the row to be updated or values in certain columns of the existing row. For more information, see [Conditional Update](#).

Example 1

In this example, several columns are updated, while a specified version of a specified column, and a specified column, are deleted.

```
private static void updateRow ( SyncClient client ,
String pkValue ) {
    // Build primary key .
    PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
    primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey primaryKey = primaryKey Builder . build
();

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

    // Update columns
    for ( int i = 0 ; i < 10 ; i ++ ) {
        rowUpdateC hange . put ( new Column ( " Col " + i
, ColumnValu e . fromLong ( i )));
    }

    // Delete the specified version of the
specified column
rowUpdateC hange . deleteColu mn ( " Col10 ",
1465373223 000L );

    // Delete specified column
rowUpdateC hange . deleteColu mns ( " Col11 " );

    client . updateRow ( new UpdateRowR equest (
rowUpdateC hange ));
}
```

Example 2

In this example, update conditions are set.

```
private static void updateRow ( SyncClient client ,
String pkValue ) {
    // Build primary key .
    PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
    primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
    PrimaryKey primaryKey = primaryKey Builder . build
();

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

    // Expect the original row to exist and
update data when the Col0 value is greater than
100
    Condition condition = new Condition ( RowExisten
ceExpectat ion . EXPECT_EXI ST );
    condition . setColumnC ondition ( new SingleColu
mnValueCon dition ( " Col0 ",
        SingleColu mnValueCon dition . CompareOpe
rator . GREATER_TH AN , ColumnValu e . fromLong ( 100 )));
}
```

```

        rowUpdateChange.setCondition(condition);

        // Update columns
        for (int i = 0; i < 10; i++) {
            rowUpdateChange.put(new Column("Col" + i,
            , ColumnValue.fromLong(i)));
        }

        // Delete the specified version of the
        specified column
        rowUpdateChange.deleteColumn("Col10",
        1465373223, 000L);

        // Delete specified column
        rowUpdateChange.deleteColumns("Col11");

        client.updateRow(new UpdateRowRequest(
        rowUpdateChange));
    }

```

DeleteRow

The DeleteRow API is used to delete a single row.

When performing DeleteRow, you can use the Conditional Update function to set conditions for the existence of the row to be deleted or values in certain columns of the existing row. For more information, see [Conditional Update](#).

Example 1

In this example, a row is deleted.

```

private static void deleteRow (SyncClient client,
String pkValue) {
    // Build primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKey
    Builder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KE
    Y_NAME, PrimaryKeyValue.fromString(pkValue));
    PrimaryKey primaryKey = primaryKeyBuilder.build
    ();

    RowDeleteChange rowDeleteChange = new
    RowDeleteChange(TABLE_NAME, primaryKey);

    client.deleteRow(new DeleteRowRequest(
    rowDeleteChange));
}

```

Example 2

In this example, deletion conditions are set.

```

private static void deleteRow (SyncClient client,
String pkValue) {
    // Build primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKey
    Builder.createPrimaryKeyBuilder();

```

```

        primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KEY_NAME ,
        PrimaryKey Value . fromString ( pkValue ));
        PrimaryKey primaryKey = primaryKey Builder . build
        ();

        RowPutChange rowPutChange = new RowPutChange
        ( TABLE_NAME , primaryKey );

        // Expect the original row to exist and
delete data when the Col0 value is greater than
100
        Condition condition = new Condition ( RowExistenceExpectation . EXPECT_EXIST );
        condition . setColumnCondition ( new SingleColumnValueCondition ( " Col0 ",
        SingleColumnValueCondition . CompareOperator . GREATER_THAN , ColumnValue . fromLong ( 100 )));
        rowDeleteChange . setCondition ( condition );

        client . deleteRow ( new DeleteRowRequest (
rowDeleteChange ));
    }

```

3.8 Condition update

The conditional update function updates table data only when specified conditions are met. If the conditions are not met, the update fails. Conditional update is applicable to PutRow, UpdateRow, DeleteRow, and BatchWriteRow.

Judgment conditions include the row existence condition and column-based condition.

- **Row existence condition:** This condition is divided into three types, 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, the update fails and the system throws an error.
- **Column-based condition:** This condition currently supports SingleColumnValueCondition and CompositeColumnValueCondition. These are used to make condition-based judgments based on the value(s) of one or more columns, similar to the conditions used by the Table Store filters.

Conditional update can be used to implement optimistic locking. When a row needs to be updated, the system first obtains the value of a column. For example, we assume Column A has a value of 1. Set the condition `Column A = 1`, update the row, and set `Column A = 2`. If the update fails, this means the row has been updated by another client.

Example 1

Construct a SingleColumnValueCondition.

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

Example 2**Construct a CompositeColumnValueCondition.**

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

Example 3

You can use Condition to implement the optimistic locking feature. The following example shows how to increment a column using Condition.

```
private static void updateRowWithCondition (
    SyncClient client, String pkValue ) {
    // Build primary key .
    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 response = client.getRow(
        new GetRowRequest(criteria));
        Row row = response.getRow();
        long col0Value = row.getLatestColumn("Col0").
        getValue().asLong();

        // Set conditional update for Col0, and use
        column value + 1.
        RowUpdateChange rowUpdateChange = new
        RowUpdateChange(TABLE_NAME, primaryKey);
        Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
        ColumnCondition columnCondition = new
        SingleColumnValueCondition("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());
        }
    }

```

3.9 Filter

Table Store filters are used to filter the results on the server side, so that the server only returns the rows or columns matching the filter conditions. Filters can be used with the `GetRow`, `BatchGetRow`, and `GetRange` APIs.

Currently, Table Store supports two filters: `SingleColumnValueFilter` and `CompositeColumnValueFilter`, which are used to determine whether to filter the data of a row based on reference column values. `SingleColumnValueFilter` only checks the value of a single reference column, whereas `CompositeColumnValueFilter` checks the values of multiple reference columns and forms a logical combination of the check results to determine whether to filter row data.

Note that the filters are used to filter the data that has been read; therefore, the reference column(s) used by `SingleColumnValueFilter` or `CompositeColumnValueFilter` must be included in the read data. If you specify the columns from which data

is read, but these columns do not include any reference columns, the filters cannot obtain reference column values. When a reference column does not exist, `SingleColumnValueFilter` uses the `passIfMissing` parameter to determine whether the filter conditions are met. This means you can select an action even when no reference column exists.

Example 1

Construct a `SingleColumnValueFilter`.

```
// Set the filter ; when the Col0 value is 0
, this row is returned .
SingleColumnValueFilter singleColumnValueFilter =
new SingleColumnValueFilter (" Col0 ",
    SingleColumnValueFilter . CompareOperator .
EQUAL , ColumnValue . fromLong ( 0 ));
// If the column Col0 does not exist , the
data is not returned .
singleColumnValueFilter . setPassIfMissing ( false );
// Only judge the latest version .
singleColumnValueFilter . setLatestVersionsOnly (
true );
```

Example 2

Construct a `CompositeColumnValueFilter`.

```
// The condition composite1 is : ( Col0 == 0 ) AND
( Col1 > 100 ).
CompositeColumnValueFilter composite1 = new
CompositeColumnValueFilter ( CompositeColumnValueFilter .
LogicOperator . AND );
SingleColumnValueFilter single1 = new SingleColumn
ValueFilter (" Col0 ",
    SingleColumnValueFilter . CompareOperator .
EQUAL , ColumnValue . fromLong ( 0 ));
SingleColumnValueFilter single2 = new SingleColumn
ValueFilter (" Col1 ",
    SingleColumnValueFilter . CompareOperator .
GREATER_THAN , ColumnValue . fromLong ( 100 ));
composite1 . addFilter ( single1 );
composite1 . addFilter ( single2 );

// The condition composite2 is : ( ( Col0 == 0 ) AND
( Col1 > 100 ) ) OR ( Col2 <= 10 ).
CompositeColumnValueFilter composite2 = new
CompositeColumnValueFilter ( CompositeColumnValueFilter .
LogicOperator . OR );
SingleColumnValueFilter single3 = new SingleColumn
ValueFilter (" Col2 ",
    SingleColumnValueFilter . CompareOperator .
LESS_EQUAL , ColumnValue . fromLong ( 10 ));
composite2 . addFilter ( composite1 );
```

```
composite2 . addFilter ( single3 );
```

3.10 Primary key column auto-increment

The auto-increment function of primary key column is a new feature launched by Table Store, and is available for Java SDK v4.2.0 and later versions.

The auto-increment function of primary key column means that if you specify a primary key column as the auto-increment column, Table Store generates a new value automatically in this column when you write data, and the generated value is the maximum value of this column under the same partition key. This feature mainly applies to system designing scenarios that need an auto-increment function applied to the primary key column, such as the item ID on e-commerce websites, user ID of large websites, post ID in forums and message ID in chat tools.

Features

- Table Store currently supports multiple primary keys, however, the first primary key is the partition key, to which auto-increment primary keys are not allowed.
- Except for the partition key, any of the other primary keys can be set to the auto-increment column.
- Because the automatic incrementation is generated on the basis of the partition key, the primary key column auto-increment function is at the partition key level.
- Only one primary key column is allowed to be set to the auto-increment column in each table for the moment.
- The automatically generated auto-increment column is of the 64-bit signed long integer type.

Interface

The primary key column auto-increment feature mainly involves two types of interfaces: creating a table and writing data.

- Create a table

When creating a table, you only need to set the attribute of the auto-increment primary key to `PrimaryKeyOption.AUTO_INCREMENT`.

Related interface: `CreateTable`

```
private static void createTable ( SyncClient client ) {  
    TableMeta tableMeta = new TableMeta ( " table_name " );
```

```
// The first column is the partition key
tableMeta . addPrimary KeyColumn ( new PrimaryKey
Schema (" PK_1 ", PrimaryKey Type . STRING ));

// The second column is the auto - increment column
with INTEGER type and the attribute is AUTO_INCRE
MENT
tableMeta . addPrimary KeyColumn ( new PrimaryKey
Schema (" PK_2 ", PrimaryKey Type . INTEGER , PrimaryKey Option
. AUTO_INCRE MENT ));

int timeToLive = - 1 ; // Never expire
int maxVersion s = 1 ; // Only one version is
saved

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

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

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

**Note:**

- The first primary key is the partition key and cannot be set as the auto-increment column.
- Only INTEGER columns can be set as the auto-increment column.
- Only one primary key auto-increment column is allowed in each table.

• Write data

When writing data, you only need to set the auto-increment column values as the placeholder `PrimaryKeyValue.AUTO_INCREMENT`.

If you want to have the primary key value automatically generated after data is written to the Table Store, you can set the `ReturnType` to `RT_PK` so that the primary key value is returned when the data is written successfully.

Related interfaces: `PutRow/UpdateRow/BatchWriteRow`

```
private static void putRow ( SyncClient client ,
String receive_id ) {
    // Construct the primary key
    PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();

    // The value in the first column is the first
four digits of md5 ( receive_id )
    primaryKey Builder . addPrimary KeyColumn ( " PK_1 ",
PrimaryKey Value . fromString ( " Hangzhou " );
}
```

```
// The third column is the primary key auto -
// increment column . This value is generated by Table
// Store . Set the auto - increment column values as the
// placeholder AUTO_INCREMENT . You do not need to
// enter a true value here .
    primaryKey Builder . addPrimary KeyColumn ( " PK_2 " ,
    PrimaryKey Value . AUTO_INCREMENT );
    PrimaryKey primaryKey = primaryKey Builder . build ();

    RowPutChange rowPutChange = new RowPutChange ( "
table_name " , primaryKey );

    // Here the return type is set to RT_PK ,
    // indicating to include the PK column value in the
    // returned result . If the ReturnType is not set , no
    // result is returned by default .
    rowPutChange . setReturnT ype ( ReturnType . RT_PK );

    // Add the attribute column and message content
    rowPutChange . addColumn ( new Column ( " content " ,
    ColumnValue . fromString ( content ) ));

    // Write data to the Table Store
    PutRowResponse response = client . putRow ( new
    PutRowRequest ( rowPutChange ));

    // Print the returned PK column
    Row returnRow = response . getRow ();
    if ( returnRow != null ) {
        System . out . println ( " PrimaryKey : " + returnRow .
        getPrimary Key (). toString ());
    }

    // Print the consumed CU
    CapacityUnit cu = response . getConsume dCapacity ().
    getCapacit yUnit ();
    System . out . println ( " Read CapacityUn it : " + cu .
    getReadCap acityUnit ());
    System . out . println ( " Write CapacityUn it : " + cu .
    getWriteCa pacityUnit ());
}
```

3.11 Error handling

Method

Currently, the Table Store Java SDK adopts the `Exception` method to handle errors. The operation is successful if the called interface does not throw an exception. If an exception is thrown, the operation fails.



Note:

Batch operation interfaces such as `BatchGetRow` and `BatchWriteRow` are successfully called only when the system checks that no exception is thrown and the status of each row is successful.

Exceptions

The Table Store Java SDK has two types of exceptions: `ClientException` and `TableStoreException`, inherited from `RuntimeException`.

- **ClientException:** Indicates an internal SDK exception, for example, an incorrect parameter value.
- **TableStoreException:** Indicates a server error generated by parsing a server error message. `TableStoreException` has the following components:
 - `getHttpStatus()`: Returned HTTP code, for example, 200 or 404.
 - `getErrorCode()`: Error type string returned by Table Store.
 - `getRequestId()`: The UUID used as a unique identifier for this request. If you cannot solve a problem, save the `RequestId` and [open a ticket](#).

3.12 Incremental data operations

Table Store provides the `ListStream` and `DescribeStream` operations for streams and `GetShardIterator` and `GetStreamRecord` operations for shards.

ListStream

`ListStream` is used to list all streams enabled for a table on the current instance.

Example

List information about all streams enabled for a table.

```
private static void listStream ( SyncClient client , String
    tableName ) {
    ListStream Request listStream Request = new ListStream
    Request ( tableName );
    ListStream Response result = client . listStream ( listStream
    Request );
}
```

DescribeStream

`DescribeStream` is used to query `creationTime`, `expirationTime`, `status`, `shards`, and the ID of the next start shard (if shards are not returned) of a stream.

Example 1

Obtain information about all shards of the current stream.

```
private static void describeStream ( SyncClient client ,
    String streamId ) {
```

```
DescribeStreamRequest desRequest = new DescribeStreamRequest ( streamId );
DescribeStreamResponse desStream = client . describeStream ( desRequest );
}
```

Example 2

Set `InclusiveStartShardId` and the maximum number of shards returned each time.

```
private static void describeStream ( SyncClient client ,
String streamId ) {
DescribeStreamRequest dsRequest = new DescribeStreamRequest ( streamId );
dsRequest . setInclusiveStartShardId ( startShardId );
dsRequest . setShardLimit ( 10 );
DescribeStreamResponse dsStream = client . describeStream ( dsRequest );
}
```

GetShardIterator

`GetShardIterator` is used to obtain the start iterator for reading a shard.

Example

Obtain the start iterator for reading a shard.

```
private static void getShardIterator ( SyncClient client ,
String streamId , String shardId ) {
GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRequest ( streamId , shardId );
GetShardIteratorResponse shardIterator = client . getShardIterator ( getShardIteratorRequest );
}
```

GetStreamRecord

`GetStreamRecord` is used to obtain each update record of a shard.

Example

Obtain the first 100 update records of a shard.

```
private static void getShardIterator ( SyncClient client ,
String shardIterator ) {
GetStreamRecordRequest streamRecordRequest = new GetStreamRecordRequest ( shardIterator );
streamRecordRequest . setLimit ( 100 );
GetStreamRecordResponse streamRecordResponse = client . getStreamRecord ( streamRecordRequest );
List < StreamRecord > records = streamRecordResponse . getRecords ();
for ( int k = 0 ; k < records . size (); k ++){
System . out . println ( " record info : " + records . get ( k ) . toString () );
}
```

```

    System.out.println("next iterator:" + streamRecordResponse.getNextStreamRecordIterator());
}

```

3.13 Multiple-row operations

The Table Store SDK provides `BatchGetRow`, `BatchWriteRow`, `GetRange`, `createRangeIterator`, and other row operation APIs.

BatchGetRow

The `BatchGetRow` API can read data from multiple rows with a single request. The parameters are the same as those of the `GetRow` API. Note that `BatchGetRow` uses the same parameter conditions for all the rows. For example, if `ColumnsToGet=[colA]`, only the `colA` value is read for all the rows.

Similar to the `BatchWriteRow` API, when using the `BatchGetRow` API, you must check the returned values. If the operation fails for some rows, the system does not throw an exception but stores information of the failed rows in `BatchGetRowResponse`. You can use the `BatchGetRowResponse#getFailedRows` method to get information on the failed rows, or use the `BatchGetRowResponse#isAllSucceed` to determine if all rows were read.

Example

Set `BatchGetRow` to read 10 rows. Set the version conditions, the columns to read, and filters.

```

private static void batchGetRow ( SyncClient client
) {
    MultiRowQueryCriteria multiRowQueryCriteria =
new MultiRowQueryCriteria ( TABLE_NAME );
    // Add 10 rows to read
    for ( int i = 0 ; i < 10 ; i ++ ) {
        PrimaryKeyBuilder primaryKeyBuilder =
PrimaryKeyBuilder.createPrimaryKeyBuilder ();
        primaryKeyBuilder.addPrimary KeyColumn (
PRIMARY_KEY_NAME , PrimaryKey Value . fromString ( " pk " + i ));
        PrimaryKey primaryKey = primaryKeyBuilder .
build ();
        multiRowQueryCriteria.addRow ( primaryKey );
    }
    // Add conditions
    multiRowQueryCriteria.setMaxVersions ( 1 );
    multiRowQueryCriteria.addColumns ToGet ( " Col0 " );
    multiRowQueryCriteria.addColumns ToGet ( " Col1 " );
    SingleColumnValueFilter singleColumnValueFilter
= new SingleColumnValueFilter ( " Col0 ",
SingleColumnValueFilter . CompareOperator .
EQUAL , ColumnValue . fromLong ( 0 ));

```

```

        singleColumnValueFilter.setPassIfMissing(false);
        multiRowQueryCriteria.setFilter(singleColumnValueFilter);

        BatchGetRowRequest batchGetRowRequest = new
        BatchGetRowRequest();
        // BatchGetRowRequest allows you to read data
        from multiple tables. A single multiRowQueryCriteria
        corresponds to a query condition for one table. You
        can add multiRowQueryCriteria.
        batchGetRowRequest.addMultiRowQueryCriteria(
        multiRowQueryCriteria);

        BatchGetRowResponse batchGetRowResponse = client
        .batchGetRow(batchGetRowRequest);

        System.out.println("Were all successful:" +
        batchGetRowResponse.isSuccess());
        if (!batchGetRowResponse.isSuccess()) {
            for (BatchGetRowResponse.RowResult rowResult
            : batchGetRowResponse.getFailedRows()) {
                System.out.println("Failed rows:" +
                batchGetRowRequest.getPrimaryKey(rowResult.getTableNames(),
                rowResult.getIndex()));
                System.out.println("Cause of failure:"
                + rowResult.getError());
            }

            /**
             * You can use the createRequestForRetry
             method to construct another request to retry failed
             rows. Here, we only construct part of the retry
             request.
             * For the retry method, we recommend
             using the SDK's custom retry policy function. This
             function allows you to retry failed rows after batch
             operations. After setting the retry policy, you do
             not need to add retry code to the calling API.
             */
            BatchGetRowRequest retryRequest = batchGetRow
            Request.createRequestForRetry(batchGetRowResponse.
            getFailedRows());
        }
    }

```

BatchWriteRow

The BatchWriteRow API allows you to perform multiple write operations with a single request. These write operations include PutRow, UpdateRow, and DeleteRow. This API also allows you to write to multiple tables at one time.

The process for constructing a single operation is the same as using the PutRow, UpdateRow, or DeleteRow API. This function also allows you to set update conditions.

When calling the BatchWriteRow API, you must be sure to check the returned values.

When calling the BatchWriteRow API, you must be sure to check the returned values.

During batch writing, some rows may be written while other rows fail. The failed row

index and error messages are inserted into the returned `BatchWriteRowResponse`. If some rows fail, the system does not throw an exception. You can use the `BatchWriteRowResponse`'s `isAllSucceed` method to judge if all rows were written successfully. Some failed operations may be ignored if you do not check for them. In some situations, the `BatchWriteRow` API may throw an exception. For example, if the server detects that some operations have incorrect parameters, the system may throw a parameter error exception. When an exception is thrown, this means that none of the operations in the request have been completed.

Example

Here, a single `BatchWriteRow` request is sent. It includes 2 `PutRow` operations, 1 `UpdateRow` operation, and 1 `DeleteRow` operation.

```
private static void batchWriteRow ( SyncClient
client ) {
    BatchWriteRowRequest batchWriteRowRequest = new
BatchWriteRowRequest ();

    // Construct rowPutChange1
    PrimaryKeyBuilder pk1Builder = PrimaryKeyBuilder
.createPrimaryKeyBuilder ();
    pk1Builder.addPrimaryKeyColumn ( PRIMARY_KEY_NAME
, PrimaryKeyValue.fromString (" pk1 "));
    RowPutChange rowPutChange1 = new RowPutChange
( TABLE_NAME , pk1Builder.build ());
    // Add several columns
    for ( int i = 0 ; i < 10 ; i ++ ) {
        rowPutChange1.addColumn ( new Column (" Col "
+ i , ColumnValue.fromLong ( i )));
    }
    // Add to batch operation
    batchWriteRowRequest.addRowChange ( rowPutChange1
);

    // Construct rowPutChange2
    PrimaryKeyBuilder pk2Builder = PrimaryKeyBuilder
.createPrimaryKeyBuilder ();
    pk2Builder.addPrimaryKeyColumn ( PRIMARY_KEY_NAME
, PrimaryKeyValue.fromString (" pk2 "));
    RowPutChange rowPutChange2 = new RowPutChange
( TABLE_NAME , pk2Builder.build ());
    // Add several columns
    for ( int i = 0 ; i < 10 ; i ++ ) {
        rowPutChange2.addColumn ( new Column (" Col "
+ i , ColumnValue.fromLong ( i )));
    }
    // Add to batch operation
    batchWriteRowRequest.addRowChange ( rowPutChange2
);

    // Construct rowUpdateChange
    PrimaryKeyBuilder pk3Builder = PrimaryKeyBuilder
.createPrimaryKeyBuilder ();
    pk3Builder.addPrimaryKeyColumn ( PRIMARY_KEY_NAME
, PrimaryKeyValue.fromString (" pk3 "));
```

```

RowUpdateC hange rowUpdateC hange = new
RowUpdateC hange ( TABLE_NAME , pk3Builder . build ());
// Add several columns
for ( int i = 0 ; i < 10 ; i ++ ) {
    rowUpdateC hange . put ( new Column ( " Col " + i
, ColumnValue . fromLong ( i )));
}
// Delete a column
rowUpdateC hange . deleteColumns ( " Col10 " );
// Add to batch operation
batchWrite RowRequest . addRowChange ( rowUpdateC
hange );

// Construct rowDeleteC hange
PrimaryKey Builder pk4Builder = PrimaryKey Builder
. createPrimaryKeyBuilder ();
pk4Builder . addPrimary KeyColumn ( PRIMARY_KEY_NAME
, PrimaryKey Value . fromString ( " pk4 " ));
RowDeleteC hange rowDeleteC hange = new
RowDeleteC hange ( TABLE_NAME , pk4Builder . build ());
// Add to batch operation
batchWrite RowRequest . addRowChange ( rowDeleteC
hange );

BatchWrite RowResponse response = client .
batchWrite Row ( batchWrite RowRequest );

System . out . println ( " Were all successful : " +
response . isSuccess ());
if (! response . isSuccess ()) {
    for ( BatchWrite RowResponse . RowResult
rowResult : response . getFailedRows ()) {
        System . out . println ( " Failed rows : " +
batchWrite RowRequest . getRowChange ( rowResult . getTableName
(), rowResult . getIndex (). getPrimary Key ());
        System . out . println ( " Cause of failure : "
+ rowResult . getError ());
    }
}
/**
 * You can use the createRequestForRetry
method to construct another request to retry failed
rows . Here , we only construct part of the retry
request .
 * For the retry method , we recommend
using the SDK ' s custom retry policy function . This
function allows you to retry failed rows after batch
operations . After setting the retry policy , you do
not need to add retry code to the calling API .
 */
BatchWrite RowRequest retryRequest =
batchWrite RowRequest . createRequestForRetry ( response .
getFailedRows ());
}
}

```

GetRange

The GetRange API is used to read data in a certain range. In Table Store tables, all rows are sorted by their primary keys and primary keys are sequentially composed

by all primary key columns. Therefore, do not assume rows are sorted by a certain column of primary keys.

The `GetRange` API allows you to read data in a forward or backward direction according to a given range. You can also limit the number of rows to read. If the range is large and the number of scanned rows or the volume of data exceeds the limit, the scan stops and the read rows and next primary key are returned. If not all rows are read, you can initiate a request to start from where the last operation left off and read the remaining rows based on the next primary key returned by the previous operation.

`GetRange` requests have the following main parameters:

- **Direction:** Enumeration type, which includes the values `FORWARD` or `BACKWARD`.
- **InclusiveStartPrimaryKey:** The primary key to start from (inclusive). If the direction is set to backward, the start primary key must be greater than the end primary key.
- **ExclusiveEndPrimaryKey:** The primary key to end on (exclusive). If the direction is set to backward, the start primary key must be greater than the end primary key.
- **Limit:** The maximum number of rows for this request.
- **ColumnsToGet:** The set of columns to read. If this is not set, all columns are read.
- **MaxVersions:** The maximum number of versions to read. At least one of the `MaxVersions` and `TimeRange` parameters must be set.
- **TimeRange:** The range of version numbers to read. At least one of the `MaxVersions` and `TimeRange` parameters must be set.
- **Filter:** The filters applied. Filters are used by the server to filter the read results again.

Example

The following operation reads in the forward direction, judges whether or not the `NextStartPrimaryKey` is null, and reads all data within the range.

```
private static void getRange ( SyncClient client ,
String startPkValue , String endPkValue ) {
    RangeRowQueryCriteria rangeRowQueryCriteria =
new RangeRowQueryCriteria ( TABLE_NAME );

    // Set StartPrimaryKey
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKey
Builder . createPrimaryKeyBuilder ();
    primaryKeyBuilder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( startPkValue ));
```

```

        rangeRowQueryCriteria a . setIncludeStartPrimaryKey ( primaryKeyBuilder . build ());

        // Set EndPrimary Key
        primaryKeyBuilder = PrimaryKeyBuilder . createPrimaryKeyBuilder ();
        primaryKeyBuilder . addPrimary KeyColumn ( PRIMARY_KEY_NAME , PrimaryKey Value . fromString ( endPkValue ));
        rangeRowQueryCriteria a . setExcludeEndPrimaryKey ( primaryKeyBuilder . build ());

        rangeRowQueryCriteria a . setMaxVersions ( 1 );

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

            // If the nextStartPrimaryKey is not null
            , continue reading .
            if ( response . getNextStartPrimaryKey () != null ) {
                rangeRowQueryCriteria a . setIncludeStartPrimaryKey ( response . getNextStartPrimaryKey ());
            } else {
                break ;
            }
        }
    }
}

```

createRangeIterator

The following operation reads data by iteration.

Example

```

private static void getRangeByIterator ( SyncClient client , String startPkValue , String endPkValue ) {
    RangeIteratorParameter rangeIteratorParameter =
    = new RangeIteratorParameter ( TABLE_NAME );

    // Set StartPrimary Key
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder . createPrimaryKeyBuilder ();
    primaryKeyBuilder . addPrimary KeyColumn ( PRIMARY_KEY_NAME , PrimaryKey Value . fromString ( startPkValue ));
    rangeIteratorParameter . setIncludeStartPrimaryKey ( primaryKeyBuilder . build ());

    // Set EndPrimary Key
    primaryKeyBuilder = PrimaryKeyBuilder . createPrimaryKeyBuilder ();
    primaryKeyBuilder . addPrimary KeyColumn ( PRIMARY_KEY_NAME , PrimaryKey Value . fromString ( endPkValue ));
    rangeIteratorParameter . setExcludeEndPrimaryKey ( primaryKeyBuilder . build ());
}

```

```
        rangeIteratorParamter.setMaxVersions ( 1 );  
        Iterator < Row > iterator = client.createRange  
eIterator ( rangeIteratorParamter );  
  
        System.out.println ( " Use Iterator to implement  
GetRange result :");  
        while ( iterator.hasNext () ) {  
            Row row = iterator.next ();  
            System.out.println ( row );  
        }  
    }
```

4 Go SDK

4.1 Preface

This document describes how to install and use Go SDK v4.0.0 and later for Table Store. It is assumed that you have activated Alibaba Cloud Table Store and created an AccessKeyId and AccessKeySecret.

- If you have not activated Alibaba Cloud Table Store or want to learn more about Table Store, visit the [Table Store homepage](#).
- If you have not created an AccessKeyId and AccessKeySecret, log on to [Alibaba Cloud AccessKey console](#) to create an AccessKey.

Download the SDK

- [SDK package](#) (including the package, source code, and samples).
- For more information about how to install the SDK package, see [Installation](#).

Version

Latest version: 4.1.0

4.2 Installation

Prerequisites

Applicable to Go SDK v1.4 and later.

Procedure

Run the following command:

```
go get github.com/alibabacloud/tablestore-go-sdk
```

4.3 Initialization

TableStoreClient is the client for Table Store. It provides callers with a series of methods for operating tables and reading/writing data from/to a single row or multiple rows.

Determine an endpoint

An endpoint is the domain name address of Alibaba Cloud Table Store in a region. It supports the following format:

Endpoint type	Description
Region address	The region of the current Table Store instance, for example, <code>https://instance.cn-hangzhou.ots.aliyuncs.com</code>

Region address of Table Store

To query the endpoint where your Table Store instance is located, follow these steps:

1. Log on to the [Table Store console](#).
2. Go to the Instance Details page and locate the instance access address, which is the endpoint of the instance.

Configure an AccessKey

To access Alibaba Cloud Table Store, you need a valid AccessKey (including an AccessKeyId and AccessKeySecret) for signature authentication. To obtain the AccessKey, follow these steps:

1. Register an [Alibaba Cloud account](#).
2. Log on to the [AccessKey console](#) to create an AccessKeyId and AccessKeySecret.

After you obtain the AccessKeyId and AccessKeySecret, initialize a TableStore.Client instance by following these steps:

Use the endpoint of Table Store to create a client.

API:

```
// Initialize the "TableStore Client" instance
// endPoint is the Table Store address ( for example ,
// "https://instance.cn-hangzhou.ots.aliyun.com:80"),
// which must start with "https://"
// accessKeyId is the AccessKeyId used to access
// Table Store .
// accessKeySecret is the AccessKeySecret used to
// access Table Store .
// instanceName is the name of the instance to
// access . You can create an instance on the Table
// Store console
func NewClient ( endPoint , instanceName , accessKeyId
, accessKeySecret string , options ... ClientOption ) *
TableStore Client
```

Example:

```
client = NewClient (" your_instance_endpoint ", " your_instance_name ",
" your_user_id ", " your_user_key ")
```

4.4 Single-row operations

The Table Store SDK provides the following single-row operation APIs: PutRow, GetRow, UpdateRow, and DeleteRow.

PutRow

Inserts data into the specified row.

API

```
// @ param PutRowRequest Encapsulate the parameters
// required to perform the PutRow operation
// @ return PutRowResponse
PutRow ( request * PutRowRequest ) (* PutRowResponse , error )
```

Example

```
putRowRequest := new ( tablestore . PutRowRequest )
putRowChannel := new ( tablestore . PutRowChannel )
putRowChannel . TableName = tableName
putPk := new ( tablestore . PrimaryKey )
putPk . AddPrimary KeyColumn ( " pk1 ", " pk1value1 ")
putPk . AddPrimary KeyColumn ( " pk2 ", int64 ( 2 ))
putPk . AddPrimary KeyColumn ( " pk3 ", [] byte ( " pk3 "))

putRowChannel . PrimaryKey = putPk
putRowChannel . AddColumn ( " col1 ", " col1data1 ")
putRowChannel . AddColumn ( " col2 ", int64 ( 3 ))
putRowChannel . AddColumn ( " col3 ", [] byte ( " test "))
putRowChannel . SetCondition ( tablestore . RowExistenceExpectation_IGNORE )
putRowRequest . PutRowChannel = putRowChannel
_, err := client . PutRow ( putRowRequest )
```

```

    if err != nil {
        fmt.Println("putrow failed with error:", err)
    } else {
        fmt.Println("putrow finished")
    }
}

```

- `RowExistenceExpectation.IGNORE` indicates that new data is still inserted no matter whether the specified row exists or not. If the inserted data is the same as the existing data, the existing data is overwritten.
- `RowExistenceExpectation.EXPECT_EXIST` indicates that new data is inserted only when the specified row exists. The existing data is overwritten.
- `RowExistenceExpectation.EXPECT_NOT_EXIST` indicates that data is inserted only when the specified row does not exist.



Note:

Obtain the full sample codes at [PutRow@GitHub](#).

GetRow

This API reads a single data row based on a given primary key.

API

```

// Return a row of data from a table
//
// @ param GetRowRequest Encapsulate the
// parameters required to perform the GetRow operation
// @ return GetRowResponse Content of the
// response to the GetRow operation
GetRow ( request * GetRowRequest ) (* GetRowResponse , error )

```

Example

Read a data row.

```

getRowRequest := new ( tablestore . GetRowRequest )
criteria := new ( tablestore . SingleRowQueryCriteria )
putPk := new ( tablestore . PrimaryKey )
putPk . AddPrimary KeyColumn ( " pk1 ", " pk1value1 ")
putPk . AddPrimary KeyColumn ( " pk2 ", int64 ( 2 ))
putPk . AddPrimary KeyColumn ( " pk3 ", [] byte ( " pk3 "))

criteria . PrimaryKey = putPk
getRowRequest . SingleRowQueryCriteria = criteria
getRowRequest . SingleRowQueryCriteria . TableName =
tableName
getRowRequest . SingleRowQueryCriteria . MaxVersion = 1
getResp , err := client . GetRow ( getRowRequest )
if err != nil {
    fmt.Println ( " getrow failed with error :", err )
} else {

```

```
fmt.Println(" get row col0 result is ", getResp .
Columns [ 0 ]. ColumnName , getResp . Columns [ 0 ]. Value ,)
}
```

**Note:**

Obtain the full sample codes at [GetRow@GitHub](#).

UpdateRow

Updates the data of the specified row. If the row does not exist, a new row is added. If the row exists, the values of the specified columns are added, modified, or deleted based on the request content.

API

```
// Update a data row in a table
// @ param UpdateRowR equest Encapsulate the
parameters required to perform the UpdateRow operation
// @ return UpdateRowR esponse Content of the response
to the UpdateRow operation
UpdateRow ( request * UpdateRowR equest ) (* UpdateRowR esponse
, error )
```

Example**Update a data row.**

```
updateRowR equest := new ( tablestore . UpdateRowR equest )
updateRowC hange := new ( tablestore . UpdateRowC hange )
updateRowC hange . TableName = tableName
updatePk := new ( tablestore . PrimaryKey )
updatePk . AddPrimary KeyColumn ( " pk1 ", " pk1value1 " )
updatePk . AddPrimary KeyColumn ( " pk2 ", int64 ( 2 ) )
updatePk . AddPrimary KeyColumn ( " pk3 ", [] byte ( " pk3 " ) )
updateRowC hange . PrimaryKey = updatePk
updateRowC hange . DeleteColumn ( " col1 " )
updateRowC hange . PutColumn ( " col2 ", int64 ( 77 ) )
updateRowC hange . PutColumn ( " col4 ", " newcol3 " )
updateRowC hange . SetCondition ( tablestore . RowExistenceExpectation_EXPECT_EXIST )
updateRowR equest . UpdateRowC hange = updateRowC hange
_, err := client . UpdateRow ( updateRowR equest )

if err != nil {
    fmt.Println ( " update failed with error :", err )
} else {
    fmt.Println ( " update row finished " )
}
```

**Note:**

Obtain the full sample codes at [UpdateRow@GitHub](#).

DeleteRow

API

```
// Delete a data row from a table
// @ param DeleteRowRequest Encapsulate the
// parameters required to perform the DeleteRow operation
// @ return DeleteRowResponse Content of the
// response to the DeleteRow operation
DeleteRow ( request * DeleteRowRequest ) (* DeleteRowResponse
, error )
```

Example

This API deletes a data row.

```
deleteRowReq := new ( tablestore . DeleteRowRequest )
deleteRowReq . DeleteRowChange = new ( tablestore .
DeleteRowChange )
deleteRowReq . DeleteRowChange . TableName = tableName
deletePk := new ( tablestore . PrimaryKey )
deletePk . AddPrimary KeyColumn ( " pk1 ", " pk1value1 " )
deletePk . AddPrimary KeyColumn ( " pk2 ", int64 ( 2 ) )
deletePk . AddPrimary KeyColumn ( " pk3 ", [] byte ( " pk3 " ) )
deleteRowReq . DeleteRowChange . PrimaryKey = deletePk
deleteRowReq . DeleteRowChange . SetCondition ( tablestore .
RowExistenceExpectation_EXPECT_EXIST )
clCondition1 := tablestore . NewSingleColumnCondition ( "
col2 ", tablestore . CT_EQUAL , int64 ( 3 ) )
deleteRowReq . DeleteRowChange . SetColumnCondition (
clCondition1 )
_, err := client . DeleteRow ( deleteRowReq )
if err != nil {
    fmt . Println ( " delete failed with error :", err )
} else {
    fmt . Println ( " delete row finished " )
}
```



Note:

Obtain the full sample codes at [DeleteRow@GitHub](#).

4.5 Multiple-row operations

The Table Store SDK provides the following multi-row operation APIs: BatchGetRow, BatchWriteRow, GetRange, and GetByIterator.

BatchGetRow

Reads several data rows in batches from one or more tables.

The BatchGetRow operation can be viewed as a set of multiple GetRow operations. Each operation is executed, results are returned, and capacity units are consumed independently.

Compared to the execution of a large number of `GetRow` operations, the `BatchGetRow` operation effectively reduces the request response time and increases the data read rate.

API

```
// Return multiple data rows in a table
//
// @ param BatchGetRow wRequest Encapsulate the parameters required to perform the BatchGetRow operation
// @ return BatchGetRow wResponse Content of the response to the BatchGetRow operation
BatchGetRow ( request * BatchGetRow wRequest ) (* BatchGetRow wResponse , error )
```

Example

Read 10 data rows in batches.

```
batchGetReq := &tablestore.BatchGetRow wRequest {}
mqCriteria := &tablestore.MultiRowQueryCriteria {}

for i := 0 ; i < 10 ; i ++ {
    pkToGet := new ( tablestore.PrimaryKey )
    pkToGet.AddPrimary KeyColumn ( " pk1 ", " pk1value1 ")
    pkToGet.AddPrimary KeyColumn ( " pk2 ", int64 ( i ))
    pkToGet.AddPrimary KeyColumn ( " pk3 ", [] byte ( " pk3 "))
    mqCriteria.AddRow ( pkToGet )
    mqCriteria.MaxVersion = 1
}
mqCriteria.TableName = tableName
batchGetReq.MultiRowQueryCriteria = append ( batchGetReq . MultiRowQueryCriteria , mqCriteria )
batchGetResponse , err := client.BatchGetRow ( batchGetReq )

if err != nil {
    fmt.Println ( " batchget failed with error :", err )
} else {
    fmt.Println ( " batchget finished ")
}
```



Note:

- `BatchGetRow` supports filtering using conditional statements.
- Obtain the full sample codes at [BatchGetRow@GitHub](#).

BatchWriteRow

Inserts, modifies, or deletes several data rows in batches in one or more tables.

It is essentially a set of multiple PutRow, UpdateRow, and DeleteRow operations. Each operation is executed, results are returned independently, and capacity units are consumed independently.

API

```
// Add , delete , or modify multiple data rows in
multiple tables
//
// @ param BatchWrite RowRequest Encapsulat e
the parameters required to perform the BatchWrite Row
operation
// @ return BatchWrite RowRespons e Content of
the response to the BatchWrite Row operation
BatchWrite Row ( request * BatchWrite RowRequest ) (* BatchWrite
RowRespons e , error )
```

Example

Write 100 data rows in batches.

```
batchWrite Req := & tablestore . BatchWrite RowRequest {}
for i := 0 ; i < 100 ; i ++ {
    putRowChan ge := new ( tablestore . PutRowChan ge )
    putRowChan ge . TableName = tableName
    putPk := new ( tablestore . PrimaryKey )
    putPk . AddPrimary KeyColumn ( " pk1 " , " pk1value1 " )
    putPk . AddPrimary KeyColumn ( " pk2 " , int64 ( i ))
    putPk . AddPrimary KeyColumn ( " pk3 " , [] byte ( " pk3 " ))
    putRowChan ge . PrimaryKey = putPk
    putRowChan ge . AddColumn ( " col1 " , " fixvalue " )
    putRowChan ge . SetCondi ti on ( tablestore . RowExisten
ceExpectat ion_IGNORE )
    batchWrite Req . AddRowChan ge ( putRowChan ge )
}

response , err := client . BatchWrite Row ( batchWrite Req )
if err != nil {
    fmt . Println ( " batch request failed with : " , response )
} else {
    fmt . Println ( " batch write row finished " )
}
```



Note:

- BatchWriteRow supports filtering using conditional statements.
- Obtain the full sample codes at [BatchWriteRow@GitHub](#).

GetRange

This API reads data within the specified primary key range.

API

```
// Read multiple data rows within the specified range
// in a table .
//
// @ param GetRangeRe quest           Encapsulate the
parameters required to perform the GetRange operation
// @ return GetRangeRe sponse         Content of the
response to the GetRange operation
GetRange ( request * GetRangeRe quest ) (* GetRangeRe sponse ,
error )
```

Example

Read data within the specified range.

```
getRangeRe quest := & tablestore . GetRangeRe quest {}
rangeRowQueryCriteria := & tablestore . RangeRowQueryCriteria {}
rangeRowQueryCriteria . TableName = tableName

startPK := new ( tablestore . PrimaryKey )
startPK . AddPrimary KeyColumnWithMinValue ( " pk1 " )
startPK . AddPrimary KeyColumnWithMinValue ( " pk2 " )
startPK . AddPrimary KeyColumnWithMinValue ( " pk3 " )
endPK := new ( tablestore . PrimaryKey )
endPK . AddPrimary KeyColumnWithMaxValue ( " pk1 " )
endPK . AddPrimary KeyColumnWithMaxValue ( " pk2 " )
endPK . AddPrimary KeyColumnWithMaxValue ( " pk3 " )
rangeRowQueryCriteria . StartPrimaryKey = startPK
rangeRowQueryCriteria . EndPrimaryKey = endPK
rangeRowQueryCriteria . Direction = tablestore . FORWARD
rangeRowQueryCriteria . MaxVersion = 1
rangeRowQueryCriteria . Limit = 10
getRangeRe quest . RangeRowQueryCriteria = rangeRowQueryCriteria

getRangeRe sp , err := client . GetRange ( getRangeRe quest )

fmt . Println ( " get range result is " , getRangeRe sp )

for ; ; {
    if err != nil {
        fmt . Println ( " get range failed with error :", err )
    }
    if ( len ( getRangeRe sp . Rows ) > 0 ) {
        for _ , row := range getRangeRe sp . Rows {
            fmt . Println ( " range get row with key " , row .
                PrimaryKey . PrimaryKeys [ 0 ] . Value , row . PrimaryKey .
                PrimaryKeys [ 1 ] . Value , row . PrimaryKey . PrimaryKeys [ 2 ] .
                Value )
        }
        if getRangeRe sp . NextStartPrimaryKey == nil {
            break
        } else {
            fmt . Println ( " next pk is :", getRangeRe sp .
                NextStartPrimaryKey . PrimaryKeys [ 0 ] . Value , getRangeRe sp .
                NextStartPrimaryKey . PrimaryKeys [ 1 ] . Value , getRangeRe
                sp . NextStartPrimaryKey . PrimaryKeys [ 2 ] . Value )
            getRangeRe quest . RangeRowQueryCriteria . StartPrimaryKey
            = getRangeRe sp . NextStartPrimaryKey
            getRangeRe sp , err = client . GetRange ( getRangeRe quest )
        }
    }
}
```

```

    }
  } else {
    break
  }

  fmt.Println("continue to query rows ")
}
fmt.Println("putrow finished ")

```

**Note:**

- GetRange supports filtering using conditional statements.
- When performing the GetRange operation, note that the data may be paged.
- Obtain the full sample codes at [GetRange@GitHub](#).

4.6 Tunnel Service-level operations

4.6.1 Installation

Download the source code package

```
go get github.com/aliyun/aliyun-tablestore-go-sdk/tunnel
```

Install dependencies

- You can go to the tunnel directory and use the dep tool to install dependencies.
 - Install the [dep](#) tool.
 - Run the dep ensure -v command.
- You can also directly run the go get command to install the dependencies:

```

go get -u go.uber.org/zap
go get github.com/cenkalti/backoff
go get github.com/golang/protobuf/proto
go get github.com/satori/go.uuid
go get github.com/stretchr/testify/assert
go get github.com/smartystrategies/goconvey/convey
go get github.com/golang/mock/gomock
go get gopkg.in/natefinch/lumberjack.v2

```

4.6.2 Quick start

Initialize the Tunnel Service client

```

// endpoint specifies the URL that is used to access
// a Table Store instance, such as https://instance.cn-
// hangzhou.ots.aliyun.com.
// instance specifies the name of an instance.

```

```
// AccessKeyId specifies the AccessKey ID that is
// used to access Table Store. AccessKeySecret specifies
// the corresponding AccessKey Secret that is used to
// access Table Store.
tunnelClient := tunnel.NewTunnelClient(endpoint, instance
    , accessKeyId, accessKeySecret)
```

Create tunnels

```
req := &tunnel.CreateTunnelRequest {
    TableName: "testTable",
    TunnelName: "testTunnel",
    Type:      tunnel.TunnelTypeBaseStream, // Set the
// type of the tunnel to differential.
}
resp, err := tunnelClient.CreateTunnel(req)
if err != nil {
    log.Fatal("create test tunnel failed", err)
}
log.Println("tunnel id is", resp.TunnelId)
```

Obtain information of existing tunnels

```
req := &tunnel.DescribeTunnelRequest {
    TableName: "testTable",
    TunnelName: "testTunnel",
}
resp, err := tunnelClient.DescribeTunnel(req)
if err != nil {
    log.Fatal("create test tunnel failed", err)
}
log.Println("tunnel id is", resp.Tunnel.TunnelId)
```

Configure a callback function and start to consume data

```
// Define a callback function.
func exampleConsumeFunction(ctx *tunnel.ChannelContext
    , records []*tunnel.Record) error {
    fmt.Println("user-defined information", ctx.
        CustomValue)
    for _, rec := range records {
        fmt.Println("tunnel record detail:", rec.String
            ())
    }
    fmt.Println("a round of records consumption
        finished")
    return nil
}

// Configure the callback function. Information about
// the callback function is passed to the SimpleProc
// essFactory API. Configure the TunnelWorkerConfig API.
workConfig := &tunnel.TunnelWorkerConfig {
    ProcessorFactory: &tunnel.SimpleProcessorFactory {
        CustomValue: "user custom interface {} value",
        ProcessFunc: exampleConsumeFunction,
    },
}
}
```

```
// Specify TunnelDaemon to continuously consume the
// specified tunnel.
daemon := tunnel.NewTunnelDaemon(tunnelClient, tunnelId,
    workConfig)
log.Fatal(daemon.Run())
```

Delete tunnels

```
req := &tunnel.DeleteTunnelRequest {
    TableName: "testTable",
    TunnelName: "testTunnel",
}
_, err := tunnelClient.DeleteTunnel(req)
if err != nil {
    log.Fatal("delete test tunnel failed", err)
}
```

4.6.3 Configuration items

Tunnel Service client

When initializing the Tunnel Service client, you can customize client configurations through the `NewTunnelClientWithConfig` API. If the `Config` parameter is not specified or is set to `nil`, `DefaultTunnelConfig` is used:

```
var DefaultTunnelConfig = &TunnelConfig {
    // Set the maximum exponential backoff value to
    // retry a failed request.
    MaxRetryElapsedTime: 45 * time.Second,
    // Set the maximum timeout period that a client
    // will wait for a response to the HTTP request.
    RequestTimeout: 30 * time.Second,
    // http.DefaultTransport
    Transport: http.DefaultTransport,
}
```

Worker

The `TunnelWorkerConfig` API contains the configurations required for a worker. The `ProcessorFactory` field is required. If other fields are unspecified, their default values are used:

```
type TunnelWorkerConfig struct {
    // The heartbeat timeout periods for the worker and
    // Tunnel Service are the same. The default value is
    // retained.
    HeartbeatTimeout time.Duration
    // Set the frequency at which worker sends heartbeat
    // information.
    HeartbeatInterval time.Duration
    // Set the API that is used to establish
    // connections for consumption of data exported through
    // Tunnel Service. The default value is retained.
    ChannelDialer ChannelDialer
}
```

```
// Generate the specific processor for the connection
// The callback function is typically used to
// initialize the SimpleProcessorFactory API .
// ProcessorFactory ChannelProcessorFactory

// Configure ZAP logs . The default value is
DefaultLog Config .
LogConfig * zap . Config
// Configure log rotation for ZAP . The default value
// is DefaultSyncer .
LogWriteSyncer zapcore . WriteSyncer
}
```

ProcessorFactory is the API that is used to configure information such as the callback function. We recommend that you use the **SimpleProcessorFactory** API that is defined in the **Tunnel Service Go SDK**:

```
type SimpleProcessorFactory struct {
    // Define and configure parameters . User - defined
    // information is passed to the ChannelContext parameter
    // contained in ProcessFunc and ShutdownFunc APIs .
    CustomValue interface {}

    // Set the interval to receive information of
    // two consecutive checkpoints . The worker records
    // the interval . In the case of CpInterval is <= 0 ,
    // DefaultCheckpointInterval is used .
    CpInterval time . Duration

    // Set the synchronous callback function for the
    // worker to process data . If an error is returned
    // from ProcessFunc , the worker sends retry requests to
    // ProcessFunc to obtain the data based on exponential
    // backoff .
    ProcessFunc func ( channelCtx * ChannelContext , records
    []* Record ) error
    // Set the synchronous callback function that is
    // used when the worker exits .
    ShutdownFunc func ( channelCtx * ChannelContext )

    // Set logs . If the Logger field is set to nil
    // , DefaultLog Config is used to initialize the Logger
    // field .
    Logger * zap . Logger
}
```

Log

Default log configurations are as follows:

```
// DefaultLog Config is used in the TunnelWorkerConfig
// and SimpleProcessorFactory APIs by default to configure
// logs .
var DefaultLog Config = zap . Config {
    Level : zap . NewAtomicLevelAt ( zap . InfoLevel ) ,
    Development : false ,
    Sampling : & zap . SamplingConfig {
        Initial : 100 ,
        Thereafter : 100 ,
    } ,
}
```

```

    Encoding : " json ",
    EncoderCon fig : zapcore . EncoderCon fig {
        TimeKey :      " ts ",
        LevelKey :      " level ",
        NameKey :       " logger ",
        CallerKey :     " caller ",
        MessageKey :    " msg ",
        Stacktrace Key : " stacktrace ",
        LineEnding :    zapcore . DefaultLin eEnding ,
        EncodeLeve l :  zapcore . LowercaseL evelEncode r ,
        EncodeTime :    zapcore . ISO8601Tim eEncoder ,
        EncodeDura tion : zapcore . SecondsDur ationEncod er ,
        EncodeCall er : zapcore . ShortCalle rEncoder ,
    },
}

```

Configurations of log rotation:

```

// DefaultSyn cer is used in the TunnelWork erConfig
// and SimpleProc essFactory APIs by default to configure
// log rotation .
var DefaultSyn cer = zapcore . AddSync (& lumberjack . Logger {
    // Set the path of logs .
    Filename : " tunnelClie nt . log ",
    // Set the maximum size of each log .
    MaxSize : 512 , // MB
    // Set the maximum number of backups that can be
    // compressed for each rotated log .
    MaxBackups : 5 ,
    // Set the maximum number of days for which logs
    // can be retained .
    MaxAge : 30 , // days
    // Specify whether to compress the rotated logs .
    Compress : true ,
})

```

4.6.4 Troubleshooting

Format definition

After Tunnel Service receives an abnormal request from a user, Tunnel Service returns the error message of an incorrect Protobuf format and an HTTP status code.

The incorrect Protobuf format is as follows:

```

message Error {
    required string code = 1 ;
    optional string message = 2 ;
    optional string tunnel_id = 3 ;
}

```

Error codes

Users of Table Store Tunnel Service Go SDKs need only to pay attention to error codes with the processing logic of "Returns an error message." Go SDKs of Table Store Tunnel Service automatically process the errors with other error codes and retry

requests. We recommend that users of Table Store Tunnel Service APIs handle errors based on the processing logic.

HTTP status code	Table Store error code	Description	Processing logic
400	OTSParameterInvalid	The error message returned when an API request parameter is incorrect or no such data table exists.	Returns an error message.
400	OTSTunnelExpired	The error message returned when the log data in an incremental or a differential tunnel expires.	Returns an error message.
403	OTSPermissionDenied	The error message returned when you are not authorized to access the specified resource.	Returns an error message.
409	OTSTunnelExist	The error message returned when the tunnel to be created already exists on the server.	Returns an error message.

HTTP status code	Table Store error code	Description	Processing logic
400	OTSSequenceNumberNotMatch	The error message returned when the serial numbers of checkpoints are inconsistent between the client and server. This error can occur when the serial numbers of checkpoints on the client are smaller than those on the server or channels compete with each other.	Use the checkpoint API to obtain the serial numbers of checkpoints again.
410	OTSResourceGone	The error message returned when a request sent to obtain the heartbeat information of the Tunnel Service client times out.	Use the tunnel ID to reconnect to Tunnel Service.
503	OTSTunnelServerUnavailable	The error message returned when an internal server error occurs.	Use exponential backoff to retry requests.

4.7 Table-level operations

4.7.1 Create tables

You can call the `CreateTable` operation to create a table based on specified table schema information.

When creating a table in Table Store, you must specify the primary key of the table. A primary key contains one to four primary key columns. Each primary key column has a name and a data type.

API

```
// Create the table based on specified table
// structure information .
// request is the instance of the CreateTable eRequest
// class , which contains tableMeta , TableOptio n , and
// ReservedTh roughput
// For more informatio n , see the documentat ion for
// the TableMeta class . After a table is created , it
// takes one minute to load the partitions .
// After that , you can perform other operations .
// Response : CreateTable eResponse
CreateTabl e ( request * CreateTable eRequest ) (* CreateTable
eResponse , error )
```

**Note:**

After the table is created in Table Store, it takes several seconds to load the table. Do not perform any operations during this period.

Examples

The following code provides an example of how to create a table with two primary key columns and a reserved read/write throughput of (0,0):

```
// Create a schema for the primary key columns . The
// schema contains the quantities , names , and types of
// the primary key columns .
// The first primary key column is the partition
// column . It is of the Integer type and named pk0 .
// The second primary key column is of the Integer
// type and named pk1 .

tableMeta := new ( tablestore . TableMeta )
tableMeta . TableName = tableName
tableMeta . AddPrimary KeyColumn ( " pk0 " , tablestore .
PrimaryKey Type_INTEG ER )
tableMeta . AddPrimary KeyColumn ( " pk1 " , tablestore .
PrimaryKey Type_STRIN G )
tableOptio n := new ( tablestore . TableOptio n )
tableOptio n . TimeToAliv e = - 1
tableOptio n . MaxVersion = 3
reservedTh roughput := new ( tablestore . ReservedTh roughput )
reservedTh roughput . Readcap = 0
reservedTh roughput . Writecap = 0
createtabl eRequest . TableMeta = tableMeta
createtabl eRequest . TableOptio n = tableOptio n
createtabl eRequest . ReservedTh roughput = reservedTh
roughput
response , err = client . CreateTabl e ( createtabl eRequest )
if ( err != nil ) {
    fmt . Println ( " Failed to create table with error
:" , err )
} else {
    fmt . Println ( " Create table finished " )
```

}

**Note:**

Code details can be obtained at [createTable@GitHub](#).

4.7.2 List table names

You can call the ListTable operation to obtain the names of all tables that are created in the current instance.

API

```
// List the names of all tables in the current
// instance. If the operation succeeds, names of all
// tables in the current instance are returned.
ListTable () (* ListTableR esponse , error )
```

Examples

The following code provides an example of how to obtain the names of all tables in the current instance:

```
tables , err := client . ListTable ()
if err != nil {
    fmt . Println ( " Failed to list table " )
} else {
    fmt . Println ( " List table result is " )
    for _ , table := range ( tables . TableNames ) {
        fmt . Println ( " TableName : " , table )
    }
}
```

**Note:**

Code details can be obtained at [listTable@GitHub](#).

4.7.3 Update tables

You can call the UpdateTable operation to update the reserved read/write throughput of a table.

API

```
// Update the values of the TableOptions and
// ReservedThroughput fields of a table.
UpdateTable ( request * UpdateTableRequest ) (* UpdateTableResponse , error )
```

Examples

The following code provides an example of how to update the configurations of a table such as setting MaxVersion to 5:

```
updateTableReq := new ( tablestore . UpdateTableRequest )
updateTableReq . TableName = tableName
updateTableReq . TableOption = new ( tablestore . TableOption )
updateTableReq . TableOption . TimeToAlive = - 1
updateTableReq . TableOption . MaxVersion = 5

_, err := client . UpdateTable ( updateTableReq )

if ( err != nil ) {
    fmt . Println ( " failed to update table with error :",
err )
} else {
    fmt . Println ( " update finished " )
}
```



Note:

Code details can be obtained at [updateTable@GitHub](#).

4.7.4 Query the description information of a table

You can call the DescribeTable operation to query the structure information and the reserved read/write throughput of a table.

API

```
// Query the descriptive information of a table by
// table name .
DescribeTable ( request * DescribeTableRequest ) (* DescribeTableResponse , error )
```

Examples

```
describeTableReq := new ( tablestore . DescribeTableRequest )
describeTableReq . TableName = tableName
describeTableReq , err := client . DescribeTable ( describeTableReq )
if err != nil {
    fmt . Println ( " failed to update table with error :",
err )
} else {
    fmt . Println ( " DescribeTableSample finished . Table meta
:", describeTableReq . TableOption . MaxVersion , describeTableReq . TableOption
. TimeToAlive )
}
```



Note:

Code details can be obtained at [describeTable@GitHub](#).

4.7.5 Delete tables

You can call the DeleteTable operation to delete a table from a specified instance.

API

DeleteTable(request *DeleteTableRequest) (*DeleteTableResponse, error)

Examples

The following code provides an example of how to delete a table from a specified instance:

```
deleteReq := new ( tablestore . DeleteTableRequest )
deleteReq . TableName = tableName
_, err := client . DeleteTable ( deleteReq )
if ( err != nil ) {
    fmt . Println ( " Failed to delete table with error :",
err )
} else {
    fmt . Println ( " Delete table finished " )
}
```



Note:

Code details can be obtained at [deleteTable@GitHub](#).

4.7.6 Create search indexes

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

You can create multiple search indexes for a table. When you create a search index, you must specify the name of the table as well as the name and schema of the search index.

Parameter description

- **TableName:** the name of the table for which you want to create a search index.
- **IndexName:** the name of the search index.
- **Schema:** specifies the index schema and its configurations.
 - **IndexSetting:** optional. **RoutingFields:** an advanced feature. You can set this parameter to customize routing fields. You can specify some primary key

columns as routing fields. Table Store distributes data that is written to a search index to different partitions based on the specified routing fields.

- **IndexSort:** optional. Index sorting is implemented based on the primary key column by default. You can also specify other columns for index sorting.
- **FieldSchemas:** the list of field schemas. The following table describes the parameters you can set for each field schema.

Name	Required	Description
FieldName	Yes	■ This parameter specifies the name of the field that is used for index sorting. ■ The field can be a primary key column or an attribute key column. ■ This parameter value is of the String type.
FieldType	Yes	The field type that is used for index sorting. For more information, see the "Supported data types (FieldType)" section in SearchIndex.
Index	No	■ This parameter specifies whether to enable the search index. ■ This parameter value is of the Boolean type.
IndexOptions	No	The configuration item for the search index.
Analyzer	No	This parameter specifies a tokenizer.
EnableSortAndAgg	No	■ This parameter specifies whether to enable index sorting and statistics collection. ■ This parameter value is of the Boolean type.
Store	No	■ This parameter specifies whether to store the field value in the search index schema. ■ This parameter value is of the Boolean type. ■ If the parameter value is true, you can read data directly from the search index schema without the need to query the table. This configuration is used to optimize query performance.

Name	Required	Description
Array	No	■ This parameter value is of the Boolean type. ■ The Boolean value specifies whether the data type is Array. ■ If this parameter value is true, the data in this column is an array. When you write data to this column, ensure that the data is a JSON array, such as ["a","b","c"]. The Nested field value is an array. You do not need to set the Array attribute for the field value.

Examples

```

/**
 * Create a search index that contains Col_Keywor d
 * and Col_Long . Set the type of data in Col_Keywor d
 * to Keyword . Set the type of data in Col_Long to
 * Long .
 */
func CreateSear chIndex ( client * tablestore . TableStore
Client , tableName string , indexName string ) {
    request := & tablestore . CreateSear chIndexReq uest {}
    request . TableName = tableName // Set the table name .
    request . IndexName = indexName // Set the search index
    name .

    schemas := []* tablestore . FieldSchem a {}
    field1 := & tablestore . FieldSchem a {
        FieldName : proto . String ( " Col_Keywor d " ), // Set the
        field name by calling the proto . String method . This
        method is used to request a string pointer .
        FieldType : tablestore . FieldType_ KEYWORD , // Set the
        field type .
        Index : proto . Bool ( true ), // Set Index to true .
        EnableSort AndAgg : proto . Bool ( true ), // Set EnableSort
        AndAgg to true to enable index sorting and statistics
        collection .
    }
    field2 := & tablestore . FieldSchem a {
        FieldName : proto . String ( " Col_Long " ),
        FieldType : tablestore . FieldType_ LONG ,
        Index : proto . Bool ( true ),
        EnableSort AndAgg : proto . Bool ( true ),
    }
    schemas = append ( schemas , field1 , field2 )

    request . IndexSchem a = & tablestore . IndexSchem a {
        FieldSchem as : schemas , // Set the fields that are
        contained in the search index .
    }
    resp , err := client . CreateSear chIndex ( request ) // Call
    a client to create the search index .
    if err != nil {
        fmt . Println ( " error : " , err )
        return
    }
}

```

```

    fmt.Println(" CreateSearchIndex finished , requestId :",
resp . ResponseInfo . RequestId )
}

```

4.7.7 List search indexes

You can call the `ListSearchIndex` operation to list all search indexes created for a table.

Examples

```

func ListSearchIndex ( client * tablestore . TableStore Client
, tableName string ) {
    request := & tablestore . ListSearchIndexRequest {}
    request . TableName = tableName // Set the table name .
    resp , err := client . ListSearchIndex ( request ) // Obtain
all search indexes created for the table .
    if err != nil {
        fmt.Println ( " error : ", err )
        return
    }
    for _ , info := range resp . IndexInfo {
        fmt.Printf ( "%#v \n " , info ) // Display the informatio
n about search indexes .
    }
    fmt.Println ( " ListSearchIndex finished , requestId :", resp
. ResponseInfo . RequestId )
}

```

4.7.8 Query descriptive information of search indexes

You can call the `DescribeSearchIndex` operation to query the descriptive information of a search index, such as its field information and index configurations.

Examples

```

func DescribeSearchIndex ( client * tablestore . TableStore
Client , tableName string , indexName string ) {
    request := & tablestore . DescribeSearchIndexRequest {}
    request . TableName = tableName // Set the table name .
    request . IndexName = indexName // Set the search index
name .
    resp , err := client . DescribeSearchIndex ( request )
    if err != nil {
        fmt.Println ( " error : ", err )
        return
    }
    fmt.Println ( " FieldSchema as :")
    for _ , schema := range resp . Schema . FieldSchema {
        fmt.Printf ( "%s \n " , schema ) // Display the schema
informatio n of the fields in the search index .
    }
    fmt.Println ( " DescribeSearchIndex finished , requestId : ",
resp . ResponseInfo . RequestId )
}

```

```
}
```

4.7.9 Delete search indexes

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

Examples

```
func DeleteSearchIndex ( client * tablestore . TableStore
Client , tableName string , indexName string ) {
    request := & tablestore . DeleteSearchIndexRequest {}
    request . TableName = tableName // Set the table name .
    request . IndexName = indexName // Set the search index
    name .
    resp , err := client . DeleteSearchIndex ( request ) // Call
    a client to delete the specified search index .

    if err != nil {
        fmt . Println ( " error : " , err )
        return
    }
    fmt . Println ( " DeleteSearchIndex finished , requestId : " ,
    resp . ResponseInfo . RequestId )
}
```

4.8 Search index-based queries

4.8.1 Query by exact match

TermQuery

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

```
/**
 * Search the Col_Keyword column of the table for
 * data that exactly matches "hangzhou."
 */
func TermQuery ( client * tablestore . TableStore Client ,
tableName string , indexName string ) {
    searchRequest := & tablestore . SearchRequest {}
    searchRequest . SetTableName ( tableName )
    searchRequest . SetIndexName ( indexName )
    query := & search . TermQuery {} // Set the query type to
    TermQuery .
    query . FieldName = " Col_Keyword " // Set the field that
    you want to match .
    query . Term = " hangzhou " // Set the value that you
    want to match .
    searchQuery := search . NewSearchQuery ()
    searchQuery . SetQuery ( query )
}
```

```

searchQuery . SetGetTotalCount ( true )
searchRequest . SetSearchQuery ( searchQuery )
// Set ColumnsToGet to true to return all columns
that match the filtering condition .
searchRequest . SetColumnsToGet (& tablestore . ColumnsToGet {
    ReturnAll : true ,
})
searchResponse , err := client . Search ( searchRequest )
if err != nil {
    fmt . Printf ("%#v\n", err )
    return
}
fmt . Println (" IsAllSuccess : ", searchResponse . IsAllSuccess ) // Check whether all data is returned .
fmt . Println (" TotalCount : ", searchResponse . TotalCount ) // Check the total number of matched rows .
fmt . Println (" RowCount : ", len ( searchResponse . Rows ))
for _ , row := range searchResponse . Rows {
    jsonBody , err := json . Marshal ( row )
    if err != nil {
        panic ( err )
    }
    fmt . Println (" Row : ", string ( jsonBody ))
}
}
}

```

TermsQuery

You can use **TermsQuery** to query data that exactly matches the specified field values . **TermQuery** is similar to **TermsQuery**. Their difference is that **TermsQuery** supports multiple terms. You can retrieve query results that match all these terms.

```

/**
 * Search the Col_Keyword column of the table for
 * data that matches " hangzhou " or " tablestore ."
 */
func TermsQuery ( client * tablestore . TableStore Client ,
    tableName string , indexName string ) {
    searchRequest := & tablestore . SearchRequest {}
    searchRequest . SetTableName ( tableName )
    searchRequest . SetIndexName ( indexName )
    query := & search . TermsQuery {} // Set the query type
    to TermsQuery .
    query . FieldName = " Col_Keyword " // Set the fields that
    you want to match .
    terms := make ([] interface {}, 0 )
    terms = append ( terms , " hangzhou " )
    terms = append ( terms , " tablestore " )
    query . Terms = terms // Set the values that you want
    to match .
    searchQuery := search . NewSearchQuery ()
    searchQuery . SetQuery ( query )
    searchQuery . SetLimit ( 100 )
    searchRequest . SetSearchQuery ( searchQuery )
    // Set ColumnsToGet to true to return all columns
    that match the filtering condition .
    searchRequest . SetColumnsToGet (& tablestore . ColumnsToGet {
        ReturnAll : true ,
    })
    searchResponse , err := client . Search ( searchRequest )
    if err != nil {

```

```

    fmt . Printf ("%# v ", err )
    return
}
fmt . Println (" IsAllSucce ss : ", searchResp onse . IsAllSucce
ss ) // Check whether all data is returned .
fmt . Println (" RowCount : ", len ( searchResp onse . Rows ))
for _ , row := range searchResp onse . Rows {
    jsonBody , err := json . Marshal ( row )
    if err != nil {
        panic ( err )
    }
    fmt . Println (" Row : ", string ( jsonBody ))
}
}
}

```

4.8.2 Query by match

MatchAllQuery

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

```

/**
 * Use MatchAllQuery to query the total number of
 * rows in a table .
 */
func MatchAllQuery ( client * tablestore . TableStore Client ,
tableName string , indexName string ) {
    searchRequest := & tablestore . SearchRequest {
        searchRequest . SetTableName ( tableName )
        searchRequest . SetIndexName ( indexName )
        query := & search . MatchAllQuery {} // Set the query type
        to MatchAllQuery .
        searchQuery := search . NewSearchQuery ()
        searchQuery . SetQuery ( query )
        searchQuery . SetGetTotalCount ( true )
        searchQuery . SetLimit ( 0 ) // Set Limit to 0 to
        indicate that no data is returned .
        searchRequest . SetSearchQuery ( searchQuery )
        searchResponse , err := client . Search ( searchRequest )
        if err != nil { // Check whether the invocation
        succeeds .
            fmt . Printf ("%# v ", err )
            return
        }
        fmt . Println (" IsAllSucce ss : ", searchResponse . IsAllSucce
ss )
        fmt . Println (" TotalCount : ", searchResponse . TotalCount
) // Check the total number of rows in the table .
    }
}

```

MatchQuery

You can use MatchQuery to query a table based on approximate matches. For example, you can query data that matches "this is." Table Store returns query results such as "..., this is tablestore," "is this tablestore," "tablestore is cool," "this," and "is."

```

/**
 * Search the Col_Keywor d column of the table for
 * data that matches " hangzhou ." Table Store returns
 * matched rows and the total number of matched rows .
 */
func MatchQuery ( client * tablestore . TableStore Client ,
tableName string , indexName string ) {
    searchRequ est := & tablestore . SearchRequ est {}
    searchRequ est . SetTableNa me ( tableName )
    searchRequ est . SetIndexNa me ( indexName )
    query := & search . MatchQuery {} // Set the query type to
    MatchQuery .
    query . FieldName = " Col_Keywor d " // Set the field that
    you want to match .
    query . Text = " hangzhou " // Set the value that you
    want to match .
    searchQuer y := search . NewSearchQ uery ()
    searchQuer y . SetQuery ( query )
    searchQuer y . SetGetTota lCount ( true )
    searchQuer y . SetOffset ( 0 ) // Set Offset to 0 .
    searchQuer y . SetLimit ( 20 ) // Set Limit to 20 to
    return up to 20 rows .
    searchRequ est . SetSearchQ uery ( searchQuer y )
    searchResp onse , err := client . Search ( searchRequ est )
    if err != nil { // Check whether the invocation
    succeeds .
        fmt . Printf ( "%# v " , err )
        return
    }
    fmt . Println ( " IsAllSucce ss : " , searchResp onse . IsAllSucce
    ss ) // Check whether all data is returned .
    fmt . Println ( " TotalCount : " , searchResp onse . TotalCount
    ) // Check the total number of matched rows .
    fmt . Println ( " RowCount : " , len ( searchResp onse . Rows ) ) //
    Check the total number of returned rows .
    for _ , row := range searchResp onse . Rows {
        jsonBody , err := json . Marshal ( row )
        if err != nil {
            panic ( err )
        }
        fmt . Println ( " Row : " , string ( jsonBody ) ) // If
        ColumnsToG et is not specified , only primary key
        columns that match the filtering condition are returned
        .
    }
    // Set ColumnsToG et to true to return all columns
    that match the filtering condition .
    searchRequ est . SetColumns ToGet (& tablestore . ColumnsToG et {
        ReturnAll : true ,
    })
    searchResp onse , err = client . Search ( searchRequ est )
    if err != nil {
        fmt . Printf ( "%# v " , err )
        return
    }
    fmt . Println ( " IsAllSucce ss : " , searchResp onse . IsAllSucce
    ss ) // Check whether all data is returned .
    fmt . Println ( " TotalCount : " , searchResp onse . TotalCount
    ) // Check the total number of matched rows .
    fmt . Println ( " RowCount : " , len ( searchResp onse . Rows ) )
    for _ , row := range searchResp onse . Rows {
        jsonBody , err := json . Marshal ( row )
        if err != nil {
            panic ( err )
        }
    }
}

```

```

    }
    fmt.Println(" Row : ", string( jsonBody ))
  }
}

```

MatchPhraseQuery

You can use MatchPhraseQuery to specify a phrase as a filtering condition.

MatchPhraseQuery is similar to MatchQuery. The difference is that MatchPhraseQuery matches a phrase as a whole. For example, if you query a phrase "this is," you can obtain query results such as "..., this is tablestore," and "this is a table." However, you cannot obtain query results such as "this table is ..." and "is this a table."

```

/**
 * Search the Col_Text column of the table for data
 * that matches " hangzhou shanghai ." Table Store returns
 * the total number of rows that match the phrase as
 * a whole and matched rows in this query .
 */
func MatchPhraseQuery ( client * tablestore . TableStore Client
, tableName string , indexName string ) {
    searchRequest := &tablestore . SearchRequest {}
    searchRequest . SetTableName ( tableName )
    searchRequest . SetIndexName ( indexName )
    query := &search . MatchPhraseQuery {} // Set the query
    type to MatchPhraseQuery .
    query . FieldName = " Col_Text " // Set the field that
    you want to match .
    query . Text = " hangzhou shanghai " // Set the value
    that you want to match .
    searchQuery := search . NewSearchQuery ()
    searchQuery . SetQuery ( query )
    searchQuery . SetGetTotalCount ( true )
    searchQuery . SetOffset ( 0 ) // Set Offset to 0 .
    searchQuery . SetLimit ( 20 ) // Set Limit to 20 to
    return up to 20 rows .
    searchRequest . SetSearchQuery ( searchQuery )
    searchResponse , err := client . Search ( searchRequest )
    if err != nil {
        fmt.Printf ("%#v ", err )
        return
    }
    fmt.Println ( " IsAllSuccess : ", searchResponse . IsAllSuccess
) // Check whether all data is returned .
    fmt.Println ( " TotalCount : ", searchResponse . TotalCount
) // Check the total number of matched rows .
    fmt.Println ( " RowCount : ", len ( searchResponse . Rows ))
    for _ , row := range searchResponse . Rows {
        jsonBody , err := json . Marshal ( row )
        if err != nil {
            panic ( err )
        }
        fmt.Println ( " Row : ", string ( jsonBody ))
    }
    // Set ColumnsToGet to true to return all columns
    that match the filtering condition .
    searchRequest . SetColumnsToGet (&tablestore . ColumnsToGet {
        ReturnAll : true ,
    })
}

```

```

searchResp onse , err = client . Search ( searchRequ est )
if err != nil {
    fmt . Printf ("%# v ", err )
    return
}
fmt . Println ( " IsAllSucce ss : ", searchResp onse . IsAllSucce
ss ) // Check whether all data is returned .
fmt . Println ( " TotalCount : ", searchResp onse . TotalCount
) // Check the total number of matched rows .
fmt . Println ( " RowCount : ", len ( searchResp onse . Rows ))
for _ , row := range searchResp onse . Rows {
    jsonBody , err := json . Marshal ( row )
    if err != nil {
        panic ( err )
    }
    fmt . Println ( " Row : ", string ( jsonBody ))
}
}
}

```

4.8.3 Query by prefix

PrefixQuery

You can use PrefixQuery to specify a prefix as a filtering condition. When a table contains a Text string, Table Store tokenizes the string and matches any of the tokens with the specified prefix.

```

/**
 * Search the Col_Keywor d column of the table for
 * data with the prefix that is an exact match of "
 * hangzhou ."
 */
func PrefixQuer y ( client * tablestore . TableStore Client ,
tableName string , indexName string ) {
    searchRequ est := & tablestore . SearchRequ est {}
    searchRequ est . SetTableNa me ( tableName )
    searchRequ est . SetIndexNa me ( indexName )
    query := & search . PrefixQuer y {} // Set the query type
    to PrefixQuer y .
    query . FieldName = " Col_Keywor d " // Set the field that
    you want to match .
    query . Prefix = " hangzhou " // Specify the prefix .
    searchQuer y := search . NewSearchQ uery ( )
    searchQuer y . SetQuery ( query )
    searchQuer y . SetGetTota lCount ( true )
    searchRequ est . SetSearchQ uery ( searchQuer y )
    // Set ColumnsToG et to true to return all columns
    that match the filtering condition .
    searchRequ est . SetColumns ToGet (& tablestore . ColumnsToG et {
        ReturnAll : true ,
    })
    searchResp onse , err := client . Search ( searchRequ est )
    if err != nil {
        fmt . Printf ("%# v ", err )
        return
    }
    fmt . Println ( " IsAllSucce ss : ", searchResp onse . IsAllSucce
ss ) // Check whether all data is returned .
    fmt . Println ( " TotalCount : ", searchResp onse . TotalCount
) // Check the total number of matched rows .
}

```

```

fmt.Println(" RowCount : ", len( searchResp onse . Rows ))
for _, row := range searchResp onse . Rows {
    jsonBody, err := json.Marshal( row )
    if err != nil {
        panic( err )
    }
    fmt.Println(" Row : ", string( jsonBody ))
}
}
}

```

4.8.4 Query by range

RangeQuery

You can use RangeQuery to specify a range as a filtering condition. When a table contains a Text string, Table Store tokenizes the string and matches any of the tokens that falls within the specified range.

```

/**
 * Search the table for rows where the value of
 * Col_Long is greater than 3. Table Store sequences
 * these rows by Col_Long in descending order.
 */
func RangeQuery( client *tablestore.TableStoreClient,
    tableName string, indexName string ) {
    searchRequest := &tablestore.SearchRequest{
        searchRequest.SetTableName( tableName )
        searchRequest.SetIndexName( indexName )
        searchQuery := search.NewSearchQuery()
        rangeQuery := &search.RangeQuery{} // Set the query type
        rangeQuery.FieldName = " Col_Long " // Set the field you
        want to match.
        rangeQuery.GT( 3 ) // Specify the range of the field
        value, which is greater than 3.
        searchQuery.SetQuery( rangeQuery )
        // Sort the query result by Col_Long in descending
        order.
        searchQuery.SetSort( &search.Sort {
            []search.Sorter {
                &search.FieldSort {
                    FieldName : " Col_Long ",
                    Order : search.SortOrder_ DESC . Enum (),
                },
            },
        })
        searchRequest.SetSearchQuery( searchQuery )
        searchRequest.SetColumnsToGet( &tablestore.ColumnsToGet {
            ReturnAll : true,
        })
        searchResponse, err := client.Search( searchRequest )
        if err != nil {
            fmt.Printf( "%#v ", err )
            return
        }
        fmt.Println( " IsAllSuccess : ", searchResponse.IsAllSuccess ) // Check whether all data is returned.
        fmt.Println( " RowCount : ", len( searchResponse.Rows ))
        for _, row := range searchResponse.Rows {
            jsonBody, err := json.Marshal( row )

```

```

    if err != nil {
        panic ( err )
    }
    fmt . Println ( " Row : ", string ( jsonBody ) )
}
}

```

4.8.5 Query by wildcard character

WildcardQuery

You can use `WildcardQuery` to match wildcard characters in a query. You can specify a value you want to match as a string that consists of one or more wildcard characters. An asterisk (*) represents any length of characters. A question mark (?) represents any single character. For example, when you specify a string "table*e" as a filtering condition, you can retrieve query results such as "tablestore." The string specified as a filtering condition cannot start with an asterisk (*).

```

/**
 * Search the Col_Keyword column of the table for
 * data that matches "hang * u ."
 */
func WildcardQuery ( client * tablestore . TableStore Client ,
    tableName string , indexName string ) {
    searchRequest := & tablestore . SearchRequest {
        searchRequest . SetTableName ( tableName )
        searchRequest . SetIndexName ( indexName )
        query := & search . WildcardQuery {} // Set the query type
        to WildcardQuery .
        query . FieldName = " Col_Keyword "
        query . Value = " hang * u "
        searchQuery := search . NewSearchQuery ()
        searchQuery . SetQuery ( query )
        searchRequest . SetSearchQuery ( searchQuery )
        // Set ColumnsToGet to true to return all columns
        that match the filtering condition .
        searchRequest . SetColumnsToGet (& tablestore . ColumnsToGet {
            ReturnAll : true ,
        })
        searchResponse , err := client . Search ( searchRequest )
        if err != nil {
            fmt . Printf ( "%# v ", err )
            return
        }
        fmt . Println ( " IsAllSuccess : ", searchResponse . IsAllSuccess ) // Check whether all data is returned .
        fmt . Println ( " TotalCount : ", searchResponse . TotalCount ) // Check the total number of matched rows .
        fmt . Println ( " RowCount : ", len ( searchResponse . Rows ) )
        for _ , row := range searchResponse . Rows {
            jsonBody , err := json . Marshal ( row )
            if err != nil {
                panic ( err )
            }
            fmt . Println ( " Row : ", string ( jsonBody ) )
        }
    }
}

```

```
}
```

4.8.6 Query by geographic location

Search index-based queries support three geographic location-based query methods: `GeoBoundingBoxQuery`, `GeoDistanceQuery`, and `GeoPolygonQuery`.

GeoBoundingBoxQuery

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

```
/**
 * The data type of Col_GeoPoint is GeoPoint. You
 * can obtain the rows where the value of Col_GeoPoint
 * falls within the range of a rectangular geographic
 * area whose top-left vertex is "10, 0" and lower-
 * right vertex is "0, 10."
 */
func GeoBoundingBoxQuery ( client * tablestore . TableStore
Client , tableName string , indexName string ) {
    searchRequest := & tablestore . SearchRequest { }
    searchRequest . SetTableName ( tableName )
    searchRequest . SetIndexName ( indexName )
    query := & search . GeoBoundingBoxQuery { } // Set the query
    type to GeoBoundingBoxQuery .
    query . FieldName = " Col_GeoPoint " // Set the field you
    want to match .
    query . TopLeft = " 10 , 0 " // Specify the coordinate s
    of the top-left vertex .
    query . BottomRight = " 0 , 10 " // Specify the coordinate
    s of the lower-right vertex .
    searchQuery := search . NewSearchQuery ( )
    searchQuery . SetQuery ( query )
    searchRequest . SetSearchQuery ( searchQuery )
    // Set ColumnsToGet to true to return all columns
    that match the filtering condition .
    searchRequest . SetColumnsToGet (& tablestore . ColumnsToGet {
        ReturnAll : true ,
    })
    searchResponse , err := client . Search ( searchRequest )
    if err != nil {
        fmt . Printf ("%#v " , err )
        return
    }
    fmt . Println ( " IsAllSuccess : " , searchResponse . IsAllSuccess )
    // Check whether all data is returned .
    fmt . Println ( " TotalCount : " , searchResponse . TotalCount )
    // Check the total number of matched rows .
    fmt . Println ( " RowCount : " , len ( searchResponse . Rows ))
    for _ , row := range searchResponse . Rows {
        jsonBody , err := json . Marshal ( row )
        if err != nil {
            panic ( err )
        }
        fmt . Println ( " Row : " , string ( jsonBody ))
    }
}
```

```
}
}
```

GeoDistanceQuery

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

```
/**
 * Search the table for rows where the value of
 * Col_GeoPoint falls within a specified distance from a
 * specified central point .
 */
func GeoDistanceQuery ( client * tablestore . TableStore Client
, tableName string , indexName string ) {
    searchRequest := & tablestore . SearchRequest {}
    searchRequest . SetTableName ( tableName )
    searchRequest . SetIndexName ( indexName )
    query := & search . GeoDistanceQuery {} // Set the query
    type to GeoDistanceQuery .
    query . FieldName = " Col_GeoPoint "
    query . CenterPoint = " 5 , 5 " // Set the coordinates
    of a central point .
    query . DistanceInMeter = 10000 . 0 // You can specify
    10 , 000 as a distance from the central point .
    searchQuery := search . NewSearchQuery ()
    searchQuery . SetQuery ( query )
    searchRequest . SetSearchQuery ( searchQuery )
    // Set ColumnsToGet to true to return all columns
    that match the filtering condition .
    searchRequest . SetColumnsToGet (& tablestore . ColumnsToGet {
        ReturnAll : true ,
    })
    searchResponse , err := client . Search ( searchRequest )
    if err != nil {
        fmt . Printf ("%# v " , err )
        return
    }
    fmt . Println ( " IsAllSuccess : " , searchResponse . IsAllSuccess ) // Check whether all data is returned .
    fmt . Println ( " TotalCount : " , searchResponse . TotalCount )
    // Check the total number of matched rows .
    fmt . Println ( " RowCount : " , len ( searchResponse . Rows ))
    for _ , row := range searchResponse . Rows {
        jsonBody , err := json . Marshal ( row )
        if err != nil {
            panic ( err )
        }
        fmt . Println ( " Row : " , string ( jsonBody ))
    }
}
```

```
}
```

GeoPolygonQuery

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

```
/**
 * Search the table for data where the value of
 * Col_GeoPoint falls within the range of the specified
 * polygon .
 */
func GeoPolygonQuery ( client * tablestore . TableStore Client
, tableName string , indexName string ) {
    searchRequest := &tablestore . SearchRequest {}
    searchRequest . SetTableName ( tableName )
    searchRequest . SetIndexName ( indexName )
    query := &search . GeoPolygonQuery {} // Set the query
    type to GeoDistanceQuery .
    query . FieldName = " Col_GeoPoint "
    query . Points = [] string { " 0 , 0 " , " 5 , 5 " , " 5 , 0 " } //
Specify the coordinate s of vertices of a polygon .
    searchQuery := search . NewSearchQuery ()
    searchQuery . SetQuery ( query )
    searchRequest . SetSearchQuery ( searchQuery )
    // Set ColumnsToGet to true to return all columns
    that match the filtering condition .
    searchRequest . SetColumnsToGet (&tablestore . ColumnsToGet {
        ReturnAll : true ,
    })
    searchResponse , err := client . Search ( searchRequest )
    if err != nil {
        fmt . Printf ("%#v " , err )
        return
    }
    fmt . Println ( " IsAllSuccess : " , searchResponse . IsAllSuccess ) // Check whether all data is returned .
    fmt . Println ( " TotalCount : " , searchResponse . TotalCount )
    // Check the total number of matched rows .
    fmt . Println ( " RowCount : " , len ( searchResponse . Rows ))
    for _ , row := range searchResponse . Rows {
        jsonBody , err := json . Marshal ( row )
        if err != nil {
            panic ( err )
        }
        fmt . Println ( " Row : " , string ( jsonBody ))
    }
}
```

4.8.7 Query by combination of conditions

BoolQuery

You can use `BoolQuery` to set a combination of filtering conditions. A `BoolQuery` contains one or more subqueries as filtering conditions. You can obtain the rows that match the filtering conditions. You can combine these subqueries in different ways

. If you specify these subqueries as `mustQueries`, Table Store returns the result that meets all filtering conditions. If you specify these subqueries as `mustNotQueries`, Table Store returns the result that meets none of the filtering conditions. You can configure the following parameters for `BoolQuery`:

```
/**
 * The document must exactly match all subqueries .
 */
List < Query > mustQueries ;
/**
 * The document must not match any subquery .
 */
List < Query > mustNotQueries ;
/**
 * The document must completely match all subfilters .
 * The filter is similar to the query . The difference
 * is that the score is not calculated if the filter
 * is used .
 */
List < Query > filterQueries ;
/**
 * The document must match the number of filtering
 * conditions that is specified by minimumShouldMatch . The
 * higher relevance results in higher scores .
 */
List < Query > shouldQueries ;
/**
 * Set minimumShouldMatch . The default value is 1 .
 */
Integer minimumShouldMatch ;
```

`BoolQuery` and Subqueries of `BoolQuery` can be a query of any type. Therefore, you can use `BoolQuery` to implement a complex combination of queries.

Examples

```
/**
 * Use BoolQuery to query data that matches a
 * combination of conditions .
 */
func BoolQuery ( client * tablestore . TableStore Client ,
  tableName string , indexName string ) {
  searchRequest := & tablestore . SearchRequest {}
  searchRequest . SetTableName ( tableName )
  searchRequest . SetIndexName ( indexName )

  /**
   * Condition 1 : Use RangeQuery to query data where
   * the value of Col_Long is greater than 3 .
   */
  rangeQuery := & search . RangeQuery {}
  rangeQuery . FieldName = " Col_Long "
  rangeQuery . GT ( 3 )

  /**
   * Condition 2 : Set MatchQuery to query data where
   * the value of Col_Keyword matches " hangzhou . "
   */
```

```

matchQuery := & search . MatchQuery {}
matchQuery . FieldName = " Col_Keywor d "
matchQuery . Text = " hangzhou "

{
/**
 * Create a BoolQuery where Condition 1 and Condition
 * 2 must be met .
 */
boolQuery := & search . BoolQuery {
    MustQuerie s : [] search . Query {
        rangeQuery ,
        matchQuery ,
    },
}
searchQuer y := search . NewSearchQ uery ()
searchQuer y . SetQuery ( boolQuery )
searchRequ est . SetSearchQ uery ( searchQuer y )
searchResp onse , err := client . Search ( searchRequ est )
if err != nil {
    fmt . Printf ("%# v ", err )
    return
}
fmt . Println ( " IsAllSucce ss : ", searchResp onse .
IsAllSucce ss ) // Check whether all data is returned .
fmt . Println ( " TotalCount : ", searchResp onse . TotalCount
) // Check the total number of matched rows .
fmt . Println ( " RowCount : ", len ( searchResp onse . Rows ))
}
{
/**
 * Create a BoolQuery where at least either of the
 * conditions must be met .
 */
boolQuery := & search . BoolQuery {
    ShouldQuer ies : [] search . Query {
        rangeQuery ,
        matchQuery ,
    },
    MinimumSho uldMatch : proto . Int32 ( 1 ),
}
searchQuer y := search . NewSearchQ uery ()
searchQuer y . SetQuery ( boolQuery )
searchRequ est . SetSearchQ uery ( searchQuer y )
searchResp onse , err := client . Search ( searchRequ est )
if err != nil {
    fmt . Printf ("%# v ", err )
    return
}
fmt . Println ( " IsAllSucce ss : ", searchResp onse .
IsAllSucce ss ) // Check whether all data is returned .
fmt . Println ( " TotalCount : ", searchResp onse . TotalCount
) // Check the total number of matched rows .
fmt . Println ( " RowCount : ", len ( searchResp onse . Rows ))
}
}
}

```

4.8.8 Query by nested field

NestedQuery

You cannot directly use a Nested field to query data. You must wrap it in a nested query. To implement the nested query, you must specify a subquery (which can be any query) and the path to the nested field.

Examples

```
/**
 * For example , the Nested columns contain nested_1 and
 * nested_2 . You need to search the col_nested . nested_1
 * column for data that matches "tablestore ."
 */
func NestedQuery ( client * tablestore . TableStore Client ,
tableName string , indexName string ) {
    searchRequest := & tablestore . SearchRequest {}
    searchRequest . SetTableName ( tableName )
    searchRequest . SetIndexName ( indexName )
    query := & search . NestedQuery { // Set the query type
to NestedQuery .
    Path : " col_nested " , // Set the path to the nested
field .
    Query : & search . TermQuery { // Create the subquery of
NestedQuery .
    FieldName : " col_nested . nested_1 " , // Set the field
that is prefixed with col_nested .
    Term : " tablestore " , // Set the value you
want to match .
    } ,
    ScoreMode : search . ScoreMode_ Avg ,
}
    searchQuery := search . NewSearchQuery ()
    searchQuery . SetQuery ( query )
    searchRequest . SetSearchQuery ( searchQuery )
    // Set ColumnsToGet to true to return all columns
that match the filtering condition .
    searchRequest . SetColumnsToGet (& tablestore . ColumnsToGet {
    ReturnAll : true ,
})
    searchResponse , err := client . Search ( searchRequest )
    if err != nil {
        fmt . Printf ("%#v " , err )
        return
    }
    fmt . Println ( " IsAllSuccess : " , searchResponse . IsAllSuccess ) // Check whether all data is returned .
    fmt . Println ( " RowCount : " , len ( searchResponse . Rows ))
    for _ , row := range searchResponse . Rows {
        jsonBody , err := json . Marshal ( row )
        if err != nil {
            panic ( err )
        }
        fmt . Println ( " Row : " , string ( jsonBody ))
    }
}
```

```
}
```

4.8.9 Index sorting and page scanning

Index sorting

The matched data is sorted based on the `IndexSort` field value when search index-based queries are used. `IndexSort` is unsuitable for sorting of nested fields. The default `IndexSort` field value is the name of the primary key column. You can define the `IndexSort` field value when you create an index. `IndexSort` determines the default return order when search index-based queries are used. If the `IndexSort` field value is not set, the result is returned based on the order of the primary key column.

Specify a sorting method

You can specify a sorting method for each query. Search index-based queries support the following sorting methods. You can also use multiple sorting methods and prioritize the methods as needed.

ScoreSort

You can use `ScoreSort` to sort the query result by score. `ScoreSort` is applicable to scenarios such as full-text indexing. Note that `ScoreSort` must be set to sort the query result by relevance. Otherwise, the query result is returned based on the `IndexSort` field value.

```
searchQuery := search.NewSearchQuery()
searchQuery.SetSort(&search.Sort{
[] search.Sorter {
&search.ScoreSort {
Order : search.SortOrder_DESC.Enum(), // Sort the
query result in descending order of scores.
},
},
})
```

PrimaryKeySort

You can use `PrimaryKeySort` to sort the query result based on the order of the primary key column.

```
searchQuery := search.NewSearchQuery()
searchQuery.SetSort(&search.Sort{
[] search.Sorter {
&search.PrimaryKeySort {
Order : search.SortOrder_ASC.Enum(),
},
},
})
```

```
})
```

FieldSort

You can use FieldSort to sort the query result based on the order of a specified column

.

```
// Sort the query result by Col_Long in descending
order .
searchQuer y . SetSort (& search . Sort {
[] search . Sorter {
& search . FieldSort {
FieldName : " Col_Long ",
Order :      search . SortOrder_  DESC . Enum (),
},
},
})
```

Prioritize columns to sort the query result.

```
searchQuer y . SetSort (& search . Sort {
[] search . Sorter {
& search . FieldSort {
FieldName : " col1 ",
Order :      search . SortOrder_  ASC . Enum (),
},
& search . FieldSort {
FieldName : " col2 ",
Order :      search . SortOrder_  DESC . Enum (),
},
},
})
```

GeoDistanceSort

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

```
searchQuer y . SetSort (& search . Sort {
[] search . Sorter {
& search . GeoDistanceSort {
FieldName : " location ", // Set the name of the
geographic al location field .
Points :      [] string {" 40 ,- 70 "}, // Set the coordinate
s of a central point .
},
},
})
```

Page scanning

Use Limit and Offset

When the total number of rows to be obtained is smaller than 2,000, you can set Limit and Offset to scan no more than 2,000 pages and return the result.

```
searchQuer y := search . NewSearchQ uery ()
searchQuer y . SetLimit ( 10 )
```

```
searchQuery . SetOffset ( 10 )
```

Use a token

If the operation does not complete reading the data that meets the filtering conditions, the server returns `NextToken`. You can use `NextToken` to continue reading the subsequent data. The sorting method cannot be set if a token is used. When the token is used, the sorting method used is the same as the previous request, regardless of the default sorting algorithm used by the system or the sorting algorithm defined by the user. Additionally, you cannot set `Offset` when a token is used. Data is scanned page by page, which results in a slow query.

```
/**
 * Use a token to read data by page .
 * If SearchResponse returns NextToken , you can use
 * this token to initiate the next query .
 * All data that matches the filtering condition is
 * returned until the NextToken field value is nil .
 */
func QueryRowsWithToken ( client * tablestore . TableStore
Client , tableName string , indexName string ) {
    querys := [] search . Query {
        & search . MatchAllQuery {},
        & search . TermQuery {
            FieldName : " Col_Keyword ",
            Term :      " tablestore ",
        },
    }
    for _ , query := range querys {
        fmt . Printf ( " Test query : %# v \n " , query )
        searchRequest := & tablestore . SearchRequest {}
        searchRequest . SetTableName ( tableName )
        searchRequest . SetIndexName ( indexName )
        searchQuery := search . NewSearchQuery ()
        searchQuery . SetQuery ( query )
        searchQuery . SetLimit ( 10 )
        searchQuery . SetGetTotalCount ( true )
        searchRequest . SetSearchQuery ( searchQuery )
        searchResponse , err := client . Search ( searchRequest )
        if err != nil {
            fmt . Printf ( "%# v " , err )
            return
        }
        rows := searchResponse . Rows
        requestCount := 1
        for searchResponse . NextToken != nil {
            searchQuery . SetToken ( searchResponse . NextToken )
            searchResponse , err = client . Search ( searchRequest )
            if err != nil {
                fmt . Printf ( "%# v " , err )
                return
            }
            requestCount ++
            for _ , r := range searchResponse . Rows {
                rows = append ( rows , r )
            }
        }
    }
}
```

```
    fmt.Println("IsAllSuccessful : ", searchResponse.IsAllSuccessful)
    fmt.Println("TotalCount : ", searchResponse.TotalCount)
    fmt.Println("RowsSize : ", len(rows))
    fmt.Println("RequestCount : ", requestCount)
}
}
```

5 NodeJS SDK

5.1 Preface

This document describes how to install and use NodeJS SDK v4.0 and later for Table Store. It is assumed that you have activated Alibaba Cloud Table Store and created an AccessKeyID and AccessKeySecret.

- If you have not activated Alibaba Cloud Table Store or want to learn more about Table Store, visit the [Table Store homepage](#).
- If you have not created an AccessKeyID and AccessKeySecret, log on to the [Alibaba Cloud AccessKey console](#) to create an AccessKey.

Download the SDK

- [SDK package](#)
- [GitHub](#)

Version

Latest version: 4.0.0

5.2 Installation

Prerequisites

Applicable to NodeJS v4.0 and later.

Procedure

Run the following command:

```
npm install tablestore
```

Example program

The Table Store NodeJS SDK provides diverse example programs for your reference or use. You can obtain the example programs using either of the following methods:

- Download and decompress the Table Store NodeJS SDK, and find the example programs in the examples folder.
- Access the [GitHub](#) project for the Table Store NodeJS.

5.3 Initialization

TableStore.Client is the client for Table Store, providing callers with a series of methods for operating tables and reading/writing data from/to a single row or multiple rows.

Determine an endpoint

An endpoint is the domain name address of Alibaba Cloud Table Store in a region. It supports the following format:

Endpoint type	Description
Region address	The region of the current Table Store instance, for example, <code>https://instance.cn-hangzhou.ots.aliyuncs.com</code>

Region address of Table Store

To query the endpoint where your Table Store instance is located, follow these steps:

1. Log on to the [Table Store console](#).
2. Go to the Instance Details page and locate the instance access address, which is the endpoint of the instance.

Configure an AccessKey

To access Alibaba Cloud Table Store, you need a valid AccessKey (including an AccessKeyId and AccessKeySecret) for signature authentication. To obtain the AccessKey, follow these steps:

1. Register an [Alibaba Cloud account](#).
2. Log on to the [AccessKey console](#) to create an AccessKeyId and AccessKeySecret.

After you obtain the AccessKeyId and AccessKeySecret, use the endpoint of Table Store to create a client and initialize a TableStore.Client instance.

Example:

```
var client = new TableStore.Client({
  accessKeyId: '< your access key id >',
  accessKeySecret: '< your access key secret >',
  endpoint: '< your endpoint >',
  instanceName: '< your instance name >',
  maxRetries: 20, // The default number of retry attempts
  is 20. You can ignore this parameter
```

```
});
```

5.4 Long type

Table Store provides five [data types](#). The relationship between the Table Store data types and the NodeJS SDK data types is as follows:

Table Store	NodeJS SDK	Description
String	string	The basic data type in JavaScript
Integer	int64	The data type for NodeJS SDK encapsulation
Double	number	The basic data type in JavaScript
Boolean	boolean	The basic data type in JavaScript
Binary	Buffer	The buffer object of NodeJS

int64 type

The Integer type of Table Store is a 64-bit signed integer, which does not have a corresponding data type in JavaScript. Therefore, a data type that can represent a 64-bit signed integer is required for NodeJS.

You can perform the following conversions.

```
var numberA = TableStore . Long . fromNumber ( 1000 );
var numberB = TableStore . Long . fromString ( ' 2000 ' );

var num = numberA . toNumber ();
    num = numberA . toString ();

var str = numberB . toNumber ();
```

```
str = numberB . toString ();
```

5.5 Table-level operations

5.5.1 Create tables

You can call the CreateTable operation to create a table based on specified table schema information.

When creating a table in Table Store, you must specify the primary key of the table. A primary key contains one to four primary key columns. Each primary key column has a name and a data type.

API

```
/**
 * Create the table based on specified table schema
 * information .
 */
createTable ( params , callback )
```



Note:

After the table is created in Table Store, it takes several seconds to load the table. Do not perform any operations during this period.

Examples

The following code provides an example of how to create a table with two primary key columns and a reserved read/write throughput of (0,0):

```
var client = require ( './ client ' );

var params = {
  tableMeta : {
    tableName : ' sampleTable ',
    primaryKey : [
      {
        name : ' gid ',
        type : ' INTEGER '
      },
      {
        name : ' uid ',
        type : ' INTEGER '
      }
    ]
  },
  reservedThroughput : {
    capacityUnit : {
      read : 0 ,
      write : 0
    }
  }
},
```

```
tableOptions : {
  timeToLive : - 1 ,// The data validity period in
seconds . - 1 indicates that the data never expires . To
set the validity period to one year , set timeToLive
to 365 × 24 × 3600 .
  maxVersions : 1 // The maximum number of versions
that can be saved in each column . The value of 1
indicates that only the latest version is saved in
each column .
}
};

client . createTable ( params , function ( err , data ) {
  if ( err ) {
    console . log ( ' error : ' , err );
    return ;
  }
  console . log ( ' success : ' , data );
});
```

**Note:**

Code details can be obtained at [createTable@GitHub](#).

5.5.2 List table names

You can call the ListTable operation to obtain the names of all tables that are created in the current instance.

API

```
/**
 * Obtain the names of all tables in the current
instance .
 */
listTable ( params , callback )
```

Examples

The following code provides an example of how to obtain the names of all tables in the current instance:

```
var client = require ( './ client ' );

client . listTable ( {}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error : ' , err );
    return ;
  }
  console . log ( ' success : ' , data );
});
```

**Note:**

Code details can be obtained at [listTable@GitHub](#).

5.5.3 Update tables

You can call the UpdateTable operation to update the maximum number of versions and reserved read/write throughput of a table.

API

```
/**
 * Update the reserved read / write throughput of a
 * specified table .
 */
updateTable ( params , callback )
```

Examples

Set maxVersions of a table to 5.

```
var client = require ( './ client ' );

var params = {
  tableName : ' sampleTable ',
  tableOptions : {
    maxVersions : 5 ,
  }
};

client . updateTable ( params , function ( err , data ) {
  if ( err ) {
    console . log ( ' error :', err );
    return ;
  }
  console . log ( ' success :', data );
});
```



Note:

Code details can be obtained at [updateTable@GitHub](#).

5.5.4 Query the description information of a table

You can call the DescribeTable operation to query the structure information and the reserved read/write throughput of a table.

API

```
/**
 * Query the structure information and the reserved
 * read / write throughput of the specified table .
 */
describeTable ( params , callback )
```

Examples

```
var client = require ( './ client ' );
```

```
var params = {
  tableName: 'sampleTable'
};

client.describeTable ( params, function ( err, data ) {
  if ( err ) {
    console.log ( 'error:', err );
    return;
  }
  console.log ( 'success:', data );
});
```

**Note:**

Code details can be obtained at [describeTable@GitHub](#).

5.5.5 Delete tables

You can call the DeleteTable operation to delete a table from a specified instance.

API

```
/**
 * Delete a table from a specified instance.
 */
deleteTable ( params, callback )
```

Examples

The following code provides an example of how to delete a table from a specified instance:

```
var client = require ( './client' );

var params = {
  tableName: 'sampleTable'
};

client.deleteTable ( params, function ( err, data ) {
  if ( err ) {
    console.log ( 'error:', err );
    return;
  }
  console.log ( 'success:', data );
});
```

**Note:**

Code details can be obtained at [deleteTable@GitHub](#).

5.5.6 Create search indexes

CreateSearchIndex

Before you use a search index to query a table, you must create a search index for the table. When you create the search index, you must specify the name of the table as well as the name and schema of the search index.

- **TableName:** the name of the table for which you want to create a search index.
- **IndexName:** the name of the search index.
- **Schema:** supports **FieldSchemas**, **IndexSetting**, and **IndexSort** parameters.
 - **FieldSchemas:** the list of field schemas. You can specify the following parameters for each field schema:
 - **FieldName:** required. This parameter specifies the name of the field that is used for index sorting. The field can be a primary key column or an attribute key column. The attribute can be a primary key column or an attribute column. This parameter value is of the String type.
 - **FieldType:** required. This parameter specifies The field type that is used for index sorting. For more information, see the "Supported data types (FieldType)" section in **SearchIndex**.
 - **isArray:** optional. This parameter value is of the Boolean type. The Boolean value specifies whether the data type is Array. If this parameter value is true , the data in this column is an array. When you write data to this column, ensure that the data is a JSON array, such as ["a","b","c"]. The Nested field value is an array. You do not need to set the Array attribute for the field value.
 - **Index:** optional. This parameter specifies whether to enable the search index. This parameter value is of the Boolean type.
 - **IndexOptions:** optional. This parameter specifies the configuration items for the search index.
 - **Analyzer:** optional. This parameter specifies a tokenizer.
 - **EnableSortAndAgg:** optional. This parameter specifies whether to enable index sorting and statistics collection. This parameter value is of the Boolean type.
 - **Store:** optional. This parameter specifies whether to store the field value in the search index schema. This parameter value is of the Boolean type. If the parameter value is true, you can read data directly from the search index

schema without the need to query the table. This configuration is used to optimize query performance.

- **IndexSetting:** optional

- **RoutingFields:** an advanced feature. You can set this parameter to customize routing fields. You can specify some primary key columns as routing fields . Table Store distributes data that is written to a search index to different partitions based on the specified routing fields.

- **IndexSort:** optional. Index sorting is implemented based on the order of primary key columns by default.

- **Sorters:** the sorting method that is suitable for the sorting of indexes. You can set **PrimaryKeySort** or **FieldSort** for this parameter.

- **PrimaryKeySort:**

- **Order:** The default value is `TableStore.SortOrder.SORT_ORDER_ASC`, which indicates that data is sorted in ascending order.

- **FieldSort:** You can customize fields for index sorting.

- **FieldName:** Set the fields for index sorting.

- **Order:** The default value is `TableStore.SortOrder.SORT_ORDER_ASC`, which indicates that data is sorted in ascending order.

Examples

```
/**
 * Create a search index that contains Col_Keyword ,
 * Col_Long , Col_Text , and Col_Nested .
 * Set the data types in these columns to Keyword ,
 * Long , Text , and Nested , respectively .
 */
client.createSearchIndex ({
  tableName : TABLE_NAME , // Set the table name .
  indexName : INDEX_NAME , // Set the search index name .
  schema : {
    fieldSchemas : [
      {
        fieldName : " Col_Keyword ",
        fieldType : TableStore . FieldType . KEYWORD , //
Set the name and type of the field .
        index : true , // Set Index to true .
        . setEnableSortAndAgg ( true ), // You can set
EnableSort AndAgg to true to enable index sorting
and statistics collection .
        store : false ,
        isArray : false
      },
      {
        fieldName : " Col_Long ",
        fieldType : TableStore . FieldType . LONG ,
```

```

        index : true ,
        enableSort AndAgg : true ,
        store : true ,
        isArray : false
    },
    {
        fieldName : " Col_Text ",
        fieldType : TableStore . FieldType . TEXT ,
        index : true ,
        enableSort AndAgg : false ,
        store : true ,
        isArray : false ,
    },
    // {
    //     fieldName : " Col_Nested ",
    //     fieldType : TableStore . FieldType . NESTED ,
    //     index : false ,
    //     enableSort AndAgg : false ,
    //     store : false ,
    //     fieldSchem as : [ // Set schemas for
Nested    fields .
    //     {
    //         fieldName : " Sub_Col_Keyword ",
    KEYWORD ,
    //         fieldType : TableStore . FieldType .
    //
    //         index : true ,
    //         enableSort AndAgg : true ,
    //         store : false ,
    //     },
    //     {
    //         fieldName : " Sub_Col_Long ",
    //         fieldType : TableStore . FieldType . LONG
    ,
    //         index : true ,
    //         enableSort AndAgg : true ,
    //         store : false ,
    //     }
    // ]
    // }
    ],
    indexSetting : { // Configure the search index .
        " routingFields ": [ " Pk_Keyword " ], // Only primary
key    columns can be set as routing fields .
        " routingPartitionSize ": null
    },
    indexSort : { // Queries that contains Nested fields
do not support index sorting .
        sorters : [
            // { // If indexSort is not specified , the
default value is PrimaryKey Sort . Data is sorted in
ascending order by default .
            //     primaryKey Sort : {
            //         order : TableStore . SortOrder .
SORT_ORDER_ASC
            //     }
            // },
            {
                fieldSort : {
                    fieldName : " Col_Keyword ",
                    order : TableStore . SortOrder . SORT_ORDER
DESC // Set the index sorting order .
                }
            }
        ]
    }
}

```

```

    }
  },
  function ( err , data ) {
    if ( err ) {
      console . log ( ' error :', err );
      return ;
    }
    console . log ( ' success :', JSON . stringify ( data , null ,
      2 ));
  });

```

5.5.7 List search indexes

ListSearchIndex

You can call the ListSearchIndex operation to list all search indexes created for a table.

Examples

```

client . listSearch Index ({
  tableName : TABLE_NAME // Set the table name .
}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error :', err );
    return ;
  }
  console . log ( ' success :', JSON . stringify ( data , null ,
    2 ));
});

```

5.5.8 Query descriptive information of search indexes

DescribeSearchIndex

You can call the DescribeSearchIndex operation to query the descriptive information of a search index, such as its field information and index configurations. **Examples**

```

client . describeSearchIndex ({
  tableName : TABLE_NAME , // Set the table name .
  indexName : INDEX_NAME // Set the search index name .
}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error :', err );
    return ;
  }
  console . log ( ' success :', JSON . stringify ( data , null ,
    2 ));
});

```

5.5.9 Delete search indexes

DeleteSearchIndex

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

```

client . deleteSearchIndex ({

```

```

        tableName : TABLE_NAME , // Set the table name .
        indexName : INDEX_NAME // Set the search index name .
    }, function ( err , data ) {
        if ( err ) {
            console . log ( ' error :', err );
            return ;
        }
        console . log ( ' success :', data );
    });

```

5.6 Single-row operations

The Table Store SDK provides the following single-row operation APIs: PutRow, GetRow, UpdateRow, and DeleteRow.

PutRow

Inserts data into the specified row.

API

```

/**
 * Insert data in the specified row . If this row
 * does not exist , a new row is added . If the row
 * exists , the original row is overwrite n .
 */
putRow ( params , callback )

```

Example

```

var TableStore = require ( '../ index . js ');
var Long = TableStore . Long ;
var client = require ( '../ client ');

var currentTimestam p = Date . now ();
var params = {
    tableName : " sampleTable ",
    condition : new TableStore . Condition ( TableStore .
RowExistenceExpectation . IGNORE , null ),
    primaryKey : [{ ' gid ': Long . fromNumber ( 20013 ) }, { ' uid
': Long . fromNumber ( 20013 ) }],
    attributeColumns : [
        { ' col1 ': ' Table Store ' },
        { ' col2 ': ' 2 ', ' timestamp ': currentTimestam p },
        { ' col3 ': ' 3 . 1 ' },
        { ' col4 ': ' - 0 . 32 ' },
        { ' col5 ': Long . fromNumber ( 123456789 ) }
    ],
    returnContent : { returnType : TableStore . ReturnType .
PrimaryKey }
};

client . putRow ( params , function ( err , data ) {
    if ( err ) {
        console . log ( ' error :', err );
        return ;
    }
}

```

```
console . log ( ' success : ' , data );
});
```

- **RowExistenceExpectation.IGNORE** indicates that new data is still inserted no matter whether the specified row exists or not. If the inserted data is the same as the existing data, the existing data is overwritten.
- **RowExistenceExpectation.EXPECT_EXIST** indicates that new data is inserted only when the specified row exists. The existing data is overwritten.
- **RowExistenceExpectation.EXPECT_NOT_EXIST** indicates that data is inserted only when the specified row does not exist.



Note:

Obtain the full sample codes at [PutRow@GitHub](#).

GetRow

This API reads a single data row based on a given primary key.

API

```
/**
 * Read a single data row based on a given
 * primary key .
 */
getRow ( params , callback )
```

Example

Read a data row.

```
var TableStore = require ( './ index . js ' );
var Long = TableStore . Long ;
var client = require ( './ client ' );

var params = {
  tableName : " sampleTabl e " ,
  primaryKey : [ { ' gid ' : Long . fromNumber ( 20004 ) } , { ' uid ' : Long . fromNumber ( 20004 ) } ] ,
  maxVersion s : 2
};
var condition = new TableStore . CompositeC ondition (
  TableStore . LogicalOpe rator . AND );
condition . addSubCond ition ( new TableStore . SingleColu
mnConditio n ( ' name ' , ' john ' , TableStore . Comparator Type .
EQUAL ));
condition . addSubCond ition ( new TableStore . SingleColu
mnConditio n ( ' addr ' , ' china ' , TableStore . Comparator Type .
EQUAL ));

params . columnFilt er = condition ;

client . getRow ( params , function ( err , data ) {
  if ( err ) {
```

```

        console . log ( ' error : ' , err ) ;
        return ;
    }
    console . log ( ' success : ' , data ) ;
});

```

**Note:**

Obtain the full sample codes at [GetRow@GitHub](#).

UpdateRow

Updates the data of the specified row. If the row does not exist, a new row is added. If the row exists, the values of the specified columns are added, modified, or deleted based on the request content.

API

```

/**
 * Update the data of the specified row . If the
 * row does not exist , a new row is added . If the
 * row exists , the values of the specified columns are
 * added , modified , or deleted based on the request
 * content .
 */
updateRow ( params , callback )

```

Example**Update a data row.**

```

var TableStore = require ( ' ../ index . js ' );
var Long = TableStore . Long ;
var client = require ( ' ./ client ' );

var params = {
    tableName : " sampleTabl e " ,
    condition : new TableStore . Condition ( TableStore .
RowExistenceExpectation . IGNORE , null ) ,
    primaryKey : [ { ' gid ' : Long . fromNumber ( 9 ) } , { ' uid ' :
Long . fromNumber ( 90 ) } ] ,
    updateOfAttributesColumns : [
        { ' PUT ' : [ { ' col4 ' : Long . fromNumber ( 4 ) } , { ' col5
' : ' 5 ' } , { ' col6 ' : Long . fromNumber ( 6 ) } ] } ,
        { ' DELETE ' : [ { ' col1 ' : Long . fromNumber ( 1496826473
186 ) } ] } ,
        { ' DELETE_ALL ' : [ ' col2 ' ] }
    ]
};

client . updateRow ( params ,
    function ( err , data ) {
        if ( err ) {
            console . log ( ' error : ' , err ) ;
            return ;
        }

        console . log ( ' success : ' , data ) ;
    }
)

```

```
});
```

**Note:**

Obtain the full sample codes at [UpdateRow@GitHub](#).

DeleteRow

API

```
/**
 * Delete a data row .
 */
deleteRow ( params , callback )
```

Example

Delete a data row.

```
var TableStore = require ('../ index . js ');
var Long = TableStore . Long ;
var client = require ('./ client ');

var params = {
  tableName : " sampleTabl e ",
  condition : new TableStore . Condition ( TableStore .
RowExisten ceExpectat ion . IGNORE , null ),
  primaryKey : [{ ' gid ': Long . fromNumber ( 8 ) }, { ' uid ':
Long . fromNumber ( 80 ) }]
};

client . deleteRow ( params , function ( err , data ) {
  if ( err ) {
    console . log (' error :', err );
    return ;
  }

  console . log (' success :', data );
});
```

**Note:**

Obtain the full sample codes at [DeleteRow@GitHub](#).

5.7 Multiple-row operations

The Table Store SDK provides the following multi-row operation APIs: BatchGetRow, BatchWriteRow, GetRange, and GetByIterator.

BatchGetRow

Reads several data rows in batches from one or more tables.

The BatchGetRow operation is essentially a set of multiple GetRow operations. Each operation is executed, results are returned, and capacity units are consumed independently.

Compared to the execution of a large number of GetRow operations, the BatchGetRow operation effectively reduces the request response time and increases the data read rate.

API

```
/**
 * Read several data rows in batches from one or
 * more tables .
 */
batchGetRow ( params , callback )
```

Example

Read multiple tables and multiple data rows in batches, and retry the operation if an error occurs in a single row.

```
var client = require ( './ client ' );
var TableStore = require ( '../ index . js ' );
var Long = TableStore . Long ;

var params = {
  tables : [{
    tableName : ' sampleTable ',
    primaryKey : [
      [ { ' gid ' : Long . fromNumber ( 20013 ) }, { ' uid ' :
Long . fromNumber ( 20013 ) } ],
      [ { ' gid ' : Long . fromNumber ( 20015 ) }, { ' uid ' :
Long . fromNumber ( 20015 ) } ]
    ],
    startColumn : " col2 ",
    endColumn : " col4 "
  },
  {
    tableName : ' notExistTable ',
    primaryKey : [
      [ { ' gid ' : Long . fromNumber ( 10001 ) }, { ' uid ' :
Long . fromNumber ( 10001 ) } ]
    ]
  }
],
};

var maxRetryTimes = 3 ;
var retryCount = 0 ;

function batchGetRow ( params ) {
  client . batchGetRow ( params , function ( err , data ) {
    if ( err ) {
      console . log ( ' error :', err );
      return ;
    }
  })
}
```

```

        var isAllSuccess = true ;
        var retryRequest = { tables : [] };
        for ( var i = 0 ; i < data . tables . length ; i ++ )
        {
            var faildRequest = { tableName : data . tables [ i
][ 0 ]. tableName , primaryKey : [] };

            for ( var j = 0 ; j < data . tables [ i ]. length
; j ++ ) {
                if ( ! data . tables [ i ][ j ]. isOk && null !
= data . tables [ i ][ j ]. primaryKey ) {
                    isAllSuccess = false ;
                    var pks = [];
                    for ( var k in data . tables [ i ][ j ].
primaryKey ) {
                        var name = data . tables [ i ][ j ].
primaryKey [ k ]. name ;
                        var value = data . tables [ i ][ j ].
primaryKey [ k ]. value ;
                        var kp = {};
                        kp [ name ] = value ;
                        pks . push ( kp );
                    }
                    faildRequest . primaryKey . push ( pks );
                } else {
                    // get success data
                }
            }

            if ( faildRequest . primaryKey . length > 0 ) {
                retryRequest . tables . push ( faildRequest );
            }
        }

        if ( ! isAllSuccess && retryCount ++ < maxRetryTimes
) {
            batchGetRow ( retryRequest );
        }

        console . log ( ' success : ' , data );
    });
}

batchGetRow ( params , maxRetryTimes );

```

**Note:**

- BatchGetRow supports filtering using conditional statements.
- Obtain the full sample codes at [BatchGetRow@GitHub](#).

BatchWriteRow

Inserts, modifies, or deletes several data rows in batches in one or more tables.

The BatchWriteRow operation is essentially a set of multiple PutRow, UpdateRow, and DeleteRow operations. Each operation is executed, results are returned, and capacity units are consumed independently.

API

```
/**
 * Modify data rows in batches .
 */
batchWrite Row ( params , callback )
```

Example

Write data rows in batches.

```
var client = require ( './ client ' );
var TableStore = require ( '../ index . js ' );
var Long = TableStore . Long ;

var params = {
  tables : [{
    tableName : ' sampleTabl e ',
    rows : [{
      type : ' PUT ',
      condition : new TableStore . Condition ( TableStore .
RowExisten ceExpectat ion . IGNORE , null ),
      primaryKey : [{ ' gid ' : Long . fromNumber ( 8 ) }, { '
uid ' : Long . fromNumber ( 80 ) } ],
      attributeC olumns : [{ ' attrCol1 ' : ' test1 ' }, { '
attrCol2 ' : ' test2 ' } ],
      returnCont ent : { returnType : TableStore .
ReturnType . Primarykey }
    } ],
  } ],
};

client . batchWrite Row ( params , function ( err , data ) {

  if ( err ) {
    console . log ( ' error : ', err );
    return ;
  }

  console . log ( ' success : ', data );
});
```



Note:

- BatchWriteRow supports filtering using conditional statements.
- Obtain the full sample codes at [BatchWriteRow@GitHub](#).

GetRange

Reads data within the specified primary key range.

API

```
/**
 * Read data within the specified primary key range .
 */
getRange ( params , callback )
```

Example

Read data within the specified range.

```
var Long = TableStore . Long ;
var client = require ( './ client ' );

var params = {
  tableName : " sampleTable ",
  direction : TableStore . Direction . FORWARD ,
  inclusiveStartPrimaryKey : [{ " gid " : TableStore . INF_MIN
}, { " uid " : TableStore . INF_MIN }],
  exclusiveEndPrimaryKey : [{ " gid " : TableStore . INF_MAX },
{ " uid " : TableStore . INF_MAX }],
  limit : 50
};

client . getRange ( params , function ( err , data ) {
  if ( err ) {
    console . log ( ' error : ', err );
    return ;
  }

  // If the value of data . next_start _primary_key is
  not empty , continue to read the data
  if ( data . next_start _primary_key ) {

  }

  console . log ( ' success : ', data );
});
```



Note:

- GetRange supports filtering using conditional statements.
- When performing the GetRange operation, note that the data may be paged.
- Obtain the full sample codes at [GetRange@GitHub](#).

5.8 Search index-based queries

5.8.1 Query by exact match

TermQuery

You can use TermQuery to query data that exactly matches the specified value of a field. When you query a Text string, Table Store tokenizes the string and exactly matches any of the tokens.

For example, Table Store tokenizes a Text string "tablestore is cool" into "tablestore," "is," and "cool." When you specify any of these tokens as a query string, you retrieve the query result that contains the token.

```
/**
 * Search the Col_Keywor d column of the table for
 * data that exactly matches " hangzhou ."
 */
client . search ({
  tableName : TABLE_NAME ,
  indexName : INDEX_NAME ,
  searchQuer y : {
    offset : 0 ,
    limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The valve of 0 indicates that no data is
returned .
    query : { // Set the query type to TermQuery .
      queryType : TableStore . QueryType . TERM_QUERY ,
      query : {
        fieldName : " Col_Keywor d " ,
        term : " hangzhou "
      }
    }
  },
  getTotalCo unt : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
},
  columnToGe t : { // Set RETURN_SPE CIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NON E to return no data .
    returnType : TableStore . ColumnRetu rnType . RETURN_ALL
  }
}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error : ' , err );
    return ;
  }
  console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ));
});
```

```
});
```

TermsQuery

You can use TermsQuery to query data that exactly matches the specified field values. TermQuery is similar to TermsQuery. The difference is that TermsQuery supports multiple terms. You can retrieve query results that match any of these terms.

```
/**
 * Search the Col_Keywor d column of the table for
 * data that matches " hangzhou " or " shanghai ."
 * TermsQuery supports multiple terms . You can retrieve
 * query results that match any of these terms .
 */
client . search ({
  tableName : TABLE_NAME ,
  indexName : INDEX_NAME ,
  searchQuer y : {
    offset : 0 ,
    limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The valve of 0 indicates that no data is
returned .
    query : { // Set the query type to TermsQuery .
      queryType : TableStore . QueryType . TERMS_QUE R Y ,
      query : {
        fieldName : " Col_Keywor d ",
        terms : [ " hangzhou ", " shanghai " ]
      }
    },
    getTotalCo unt : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
  },
  columnToGe t : { // Set RETURN_SPE CIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NON E to return no data .
    returnType : TableStore . ColumnRetu rnType . RETURN_ALL
  }
}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error : ' , err );
    return ;
  }
  console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ) );
});
```

```
});
```

5.8.2 Query by match

MatchAllQuery

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

```
/**
 * Use MatchAllQuery to query the total number of
 * rows in a table .
 */
client . search ({
  tableName : TABLE_NAME ,
  indexName : INDEX_NAME ,
  searchQuery : {
    offset : 0 ,
    limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The value of 0 indicates that no data is
returned .
    query : {
      queryType : TableStore . QueryType . MATCH_ALL_ QUERY
    },
    getTotalCount : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
  },
  columnToGet : { // Set RETURN_SPECIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NONE to return no data .
    returnType : TableStore . ColumnReturnType . RETURN_SPE
CIFIED ,
    returnNames : [ " Col_1 " , " Col_2 " , " Col_3 " ]
  }
}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error : ' , err );
    return ;
  }
  console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ));
});
```

MatchQuery

You can use MatchQuery to query a table based on approximate matches. For example, you can query data that matches "this is." Table Store returns query results such as "..., this is tablestore," "is this tablestore," "tablestore is cool," "this," and "is."

```
/**
 * Search the Col_Keyword column of the table for
 * data that matches " hangzhou ." Table Store returns
 * matched rows and the total number of matched rows .
```

```

*/
client . search ({
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuery : {
        offset : 0 ,
        limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The value of 0 indicates that no data is
returned .
        query : { // Set the query type to MatchQuery .
            queryType : TableStore . QueryType . MATCH_QUERY ,
            query : {
                fieldName : " Col_Keyword ",
                text : " hangzhou " // Set the value that
you want to match .
            }
        },
        getTotalCount : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
    },
    columnToGet : { // Set RETURN_SPECIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NONE to return no data .
        returnType : TableStore . ColumnReturnType . RETURN_ALL
    }
}, function ( err , data ) {
    if ( err ) {
        console . log ( ' error : ' , err );
        return ;
    }
    console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ));
});

```

MatchPhraseQuery

You can use MatchPhraseQuery to specify a phrase as a filtering condition.

MatchPhraseQuery is similar to MatchQuery. The difference is that MatchPhraseQuery matches the exact phrase as a whole. For example, if you query a phrase "this is," you can obtain query results such as "..., this is tablestore," and "this is a table." However, you cannot obtain query results such as "this table is ..." and "is this a table."

```

/**
 * Search the Col_Text column of the table for data
that matches " hangzhou shanghai ."
 * Table Store returns the total number of rows that
match the phrase as a whole and matched rows in
this query .
 */
client . search ({
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuery : {
        offset : 0 ,

```

```

        limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The value of 0 indicates that no data is
returned .
        query : { // Set the query type to MatchPhras
eQuery .
            queryType : TableStore . QueryType . MATCH_PHRA
SE_QUERY ,
            query : {
                fieldName : " Col_Text " , // Set the field
that you want to match .
                text : " hangzhou shanghai " // Set the value
that you want to match .
            }
        } ,
        getTotalCo unt : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
    } ,
    columnToGe t : { // Set RETURN_SPE CIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NON E to return no data .
        returnType : TableStore . ColumnRetu rnType . RETURN_ALL
    }
} , function ( err , data ) {
    if ( err ) {
        console . log ( ' error : ' , err ) ;
        return ;
    }
    console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ) ) ;
} ) ;
});

```

5.8.3 Query by prefix

PrefixQuery

You can use PrefixQuery to specify a prefix as a filtering condition. When a table contains a Text string, Table Store tokenizes the string and matches any of the tokens with the specified prefix.

Examples

```

/**
 * Search the Col_Keywor d column of the table for
data with the prefix that is an exact match of "
hang ," such as " hangzhou ."
 */
client . search ( {
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuer y : {
        offset : 0 ,
        limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of

```

```

    rows . The valve of 0 indicates that no data is
    returned .
    query : { // Set the query type to PrefixQuery .
      queryType : TableStore . QueryType . PREFIX_QUERY ,
      query : {
        fieldName : " Col_Keyword ",
        prefix : " hang " // Specify the prefix , such
        as " hangzhou " or " hangzhoushi ."
      }
    },
    getTotalCount : true // If TotalCount is set
    to true , the total number of rows that meet the
    filtering conditions is returned . If TotalCount is set
    to false , the total number of rows that meet the
    filtering conditions is not returned .
  },
  columnToGet : { // Set RETURN_SPECIFIED to return a
  specified number of columns , RETURN_ALL to return all
  columns , or RETURN_NONE to return no data .
    returnType : TableStore . ColumnRetur nType . RETURN_ALL
  }
}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error : ' , err );
    return ;
  }
  console . log ( ' success : ' , JSON . stringify ( data , null ,
  2 ));
});

```

5.8.4 Query by range

RangeQuery

You can use RangeQuery to specify a range as a filtering condition. When a table contains a Text string, Table Store tokenizes the string and matches any of the tokens that falls within the specified range.

Examples

```

/**
 * Search the table for rows where the value range
 * of Col_Long is [ 1 , 10 ). Table Store sorts these
 * rows by Col_Long in descending order .
 */
client . search ({
  tableName : TABLE_NAME ,
  indexName : INDEX_NAME ,
  searchQuery : {
    offset : 0 ,
    limit : 10 , // You can set Limit to 0 if you
    do not need the specific data but the number of
    rows . The valve of 0 indicates that no data is
    returned .
    query : { // Set the query type to RangeQuery .
      queryType : TableStore . QueryType . RANGE_QUERY ,
      query : {
        fieldName : " Col_Long ",
        rangeFrom : 1 ,

```

```

        includeLower : true , // >= 1
        rangeTo : 10 ,
        includeUpper : false // < 10
    },
    },
    getTotalCount : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
    },
    columnToGet : { // Set RETURN_SPECIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NONE to return no data .
        returnType : TableStore . ColumnReturnType . RETURN_ALL
    }
}, function ( err , data ) {
    if ( err ) {
        console . log ( ' error :', err );
        return ;
    }
    console . log ( ' success :', JSON . stringify ( data , null ,
2 ));
});

```

5.8.5 Query by wildcard character

WildcardQuery

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

An asterisk (*) represents any length of characters. A question mark (?) represents any single character. For example, when you specify a string "table*e" as a filtering condition, you can retrieve query results such as "tablestore." The string specified as a filtering condition cannot start with an asterisk (*).

Examples

```

/**
 * Search the Col_Keyword column of the table for
data that matches "hang * u ."
 */
client . search ({
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuery : {
        offset : 0 ,
        limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The value of 0 indicates that no data is
returned .
        query : { // Set the query type to WildcardQuery
.
            queryType : TableStore . QueryType . WILDCARD_QUERY ,
            query : {

```

```

        fieldName : " Col_Keywor d ",
        value : " table * e " // Set wildcard characters
    },
    },
    getTotalCount : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
    },
    columnToGet : { // Set RETURN_SPECIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NONE to return no data .
        returnType : TableStore . ColumnRetu rnType . RETURN_ALL
    }
}, function ( err , data ) {
    if ( err ) {
        console . log ( ' error : ' , err );
        return ;
    }
    console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ));
});

```

5.8.6 Query by geographic location

Search index-based queries support three geographic location-based query methods: GeoBoundingBoxQuery, GeoDistanceQuery, and GeoPolygonQuery.

GeoBoundingBoxQuery

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

Examples

```

/**
 * The data type of Col_GeoPoi nt is GeoPoint .
 * You can obtain the rows that meet the following
 * conditions :
 * The value of Col_GeoPoi nt falls within the
 * range of a rectangular geographic al area whose
 * coordinate s of the top - left vertex are " 10 , 0 " and
 * coordinate s of the lower - right vertex are " 0 , 10
 * ."
 */
client . search ({
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuer y : {
        offset : 0 ,
        limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The valve of 0 indicates that no data is
returned .
    }
})

```

```

        query : { // Set the query type to GeoBoundin
gBoxQuery .
        queryType : TableStore . QueryType . GEO_BOUNDI
NG_BOX_QUE
RY ,
        query : {
            fieldName : " Col_GeoPoi nt ", // Set the field
you want to match .
            topLeft : " 10 , 0 ", // Specify the coordinate
s of the top - left vertex .
            bottomRigh t : " 0 , 10 " // Specify the
coordinate s of the lower - right vertex .
        }
    },
    getTotalCo unt : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
    },
    columnToGe t : { // Set RETURN_SPE CIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NON E to return no data .
        returnType : TableStore . ColumnRetu rnType . RETURN_ALL
    }
}, function ( err , data ) {
    if ( err ) {
        console . log ( ' error : ' , err );
        return ;
    }
    console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ));
});

```

GeoDistanceQuery

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

Examples

```

/**
 * Search the table for rows where the value of
Col_GeoPoi nt falls within a specified distance from a
specified central point .
 */
client . search ({
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuer y : {
        offset : 0 ,
        limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The valve of 0 indicates that no data is
returned .
        query : { // Set the query type to GeoDistanc
eQuery .
            queryType : TableStore . QueryType . GEO_DISTAN
CE_QUERY ,

```

```

        query : {
            fieldName : " Col_GeoPoi nt ",
            centerPoin t : " 1 , 1 ", // Set the
coordinate s of a central point .
            distance : 10000 // You can specify 10 , 000
as a distance from the central point .
        }
    },
    getTotalCo unt : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
    },
    columnToGe t : { // Set RETURN_SPE CIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NON E to return no data .
        returnType : TableStore . ColumnRetu rnType . RETURN_ALL
    }
}, function ( err , data ) {
    if ( err ) {
        console . log ( ' error : ' , err );
        return ;
    }
    console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ) );
});

```

GeoPolygonQuery

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

Examples

```

/**
 * Search the table for data where the value of
Col_GeoPoi nt falls within the range of the specified
polygon .
 */
client . search ( {
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuer y : {
        offset : 0 ,
        limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The valve of 0 indicates that no data is
returned .
        query : { // Set the query type to GeoPolygon
Query .
            queryType : TableStore . QueryType . GEO_POLYGO N_QUERY
        ,
            query : {
                fieldName : " Col_GeoPoi nt ",
                points : [ " 0 , 0 " , " 5 , 5 " , " 5 , 0 " ] // Specify
the coordinate s of vertices of a polygon .
            }
        }
    }
},

```

```

        getTotalCount : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
    },
    columnToGet : { // Set RETURN_SPECIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NONE to return no data .
        returnType : TableStore . ColumnReturnType . RETURN_ALL
    }
}, function ( err , data ) {
    if ( err ) {
        console . log ( ' error :', err );
        return ;
    }
    console . log ( ' success :', JSON . stringify ( data , null ,
2 )));
});

```

5.8.7 Query by combination of conditions

BoolQuery

You can use BoolQuery to set a combination of filtering conditions. A BoolQuery contains one or more subqueries as filtering conditions. You can obtain the rows that match the filtering conditions.

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

Examples

```

/**
 * Use BoolQuery to query data that matches a
 * combination of conditions . Parameter description
 * mustQueries : Condition 1 and Condition 2 must be
 * met at the same time ( A AND B ).
 * shouldQueries : Either Condition 1 or Condition 2
 * is met , or both of them must be met ( A OR B ).
 * filterQueries : The document must completely match
 * all subfilters ( A AND B ). filterQueries are similar
 * to mustQueries . The difference is that the scores
 * are not calculated if filterQueries are used .
 * mustNotQueries : Neither Condition 1 nor Condition 2
 * is met (! A AND ! B ).
 * minimumShouldMatch : The minimum number of shouldQuer
 * ies that must be met . The default value is 1 .
 */
client . search ({
    tableName : TABLE_NAME ,
    indexName : INDEX_NAME ,
    searchQuery : {
        offset : 0 ,

```

```

        limit : 10 , // You can set Limit to 0 if you
        do not need the specific data but the number of
        rows . The valve of 0 indicates that no data is
        returned .
        query : { // 6
            queryType : TableStore . QueryType . BOOL_QUERY ,
            query : {
                mustQuerie s : [ // mustQuerie s , shouldQuer ies
                , and mustNotQue ries are optional .
                { // Condition 1 : Use RangeQuery to
                query data where the value of Col_Long is greater
                than 3 .
                    queryType : TableStore . QueryType .
                    RANGE_QUER Y ,
                    query : {
                        fieldName : " Col_Long ",
                        rangeFrom : 3 ,
                        includeLow er : true
                    }
                },
                { // Condition 2 : Set MatchQuery to
                query data where the value of Col_Keywor d matches "
                hangzhou ."
                    queryType : TableStore . QueryType .
                    TERM_QUERY ,
                    query : {
                        fieldName : " Col_Keywor d ",
                        term : " hangzhou "
                    }
                }
            ],
            minimumSho uldMatch : 1 // Set minimumSho
            uldMatch to 1 .
        },
        getTotalCo unt : true // If TotalCount is set
        to true , the total number of rows that meet the
        filtering conditions is returned . If TotalCount is set
        to false , the total number of rows that meet the
        filtering conditions is not returned .
    },
    columnToGe t : { // Set RETURN_SPE CIFIED to return a
    specified number of columns , RETURN_ALL to return all
    columns , or RETURN_NON E to return no data .
        returnType : TableStore . ColumnRetu rnType . RETURN_ALL
    }
}, function ( err , data ) {
    if ( err ) {
        console . log ( ' error : ' , err );
        return ;
    }
    console . log ( ' success : ' , JSON . stringify ( data , null ,
    2 ));
}

```

```
});
```

5.8.8 Query by nested field

NestedQuery

You cannot directly use a Nested field to query data. You must wrap the Nested field in a nested query. To implement the nested query, you must specify a subquery (which can be any query) and the path of the nested field.

Examples

```
/**
 * Search the Col_Nested.Sub_Col_Keyword column for
 * data that matches "hangzhou."
 * Col_Nested: '[{ Sub_Col_Keyword : " Hangzhou "},{ Sub_Col_Keyword : " Shanghai "}]'
 */
client . search ({
  tableName : TABLE_NAME ,
  indexName : INDEX_NAME ,
  searchQuery : {
    offset : 0 ,
    limit : 10 , // You can set Limit to 0 if you
do not need the specific data but the number of
rows . The value of 0 indicates that no data is
returned .
    query : { // Set the query type to NestedQuery .
      queryType : TableStore . QueryType . NESTED_QUERY ,
      query : {
        path : " Col_Nested ",
        query : {
          queryType : TableStore . QueryType . MATCH_ALL_
QUERY ,
          query : {
            fieldName : " Col_Nested . Sub_Col_Keyword
",
            term : " Hangzhou "
          }
        }
      }
    },
    getTotalCount : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
  },
  columnGet : { // Set RETURN_SPECIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NONE to return no data .
    returnType : TableStore . ColumnRetur nType . RETURN_ALL
  }
}, function ( err , data ) {
  if ( err ) {
    console . log ( ' error :', err );
    return ;
  }
  console . log ( ' success :', JSON . stringify ( data , null ,
2 ));
```

```
});
```

5.8.9 Index sorting and page scanning

Index sorting

The matched data is sorted based on the IndexSort field value when search index-based queries are used. IndexSort does not support nested queries. The default IndexSort field is the name of the primary key column. You can define the IndexSort field when you create an index. IndexSort determines the default return order when search index-based queries are used. If the IndexSort field is not set, the result is returned based on the order of primary key columns.

Specify a sorting method

You can specify a sorting method for each query. Search index-based queries support the following sorting methods. You can also use multiple sorting methods and prioritize each method as needed.

ScoreSort

You can use ScoreSort to sort the query result by score. ScoreSort is applicable to scenarios such as full-text indexing. Note: ScoreSort must be set to sort the query result by relevance. Otherwise, the query result is returned based on the IndexSort field. Set index sorting fields:

```
{
  sorters : [
    {
      scoreSort : {
        order : TableStore . SortOrder . SORT_ORDER _ASC
      }
    }
  ]
}
```

PrimaryKeySort

You can use PrimaryKeySort to sort the query result based on the order of primary key columns. Set index sorting fields:

```
{
  sorters : [
    {
      primaryKey Sort : {
        order : TableStore . SortOrder . SORT_ORDER _DESC
        // The query result is sorted in descending order .
        // order : TableStore . SortOrder . SORT_ORDER _ASC
        // The query result is sorted in ascending order .
      }
    }
  ]
}
```

```

    }
  ]
}

```

FieldSort

You can use FieldSort to sort the query result by column. Set index sorting fields:

```

{
  sorters : [
    {
      fieldSort : {
        fieldName : " Col_Keywor d ",
        order : TableStore . SortOrder . SORT_ORDER _DESC
      }
    }
  ]
}

```

Prioritize columns to sort the query result. Set index sorting fields:

```

{
  sorters : [
    {
      fieldSort : {
        fieldName : " Col_Keywor d ",
        order : TableStore . SortOrder . SORT_ORDER _DESC
      }
    },
    {
      fieldSort : {
        fieldName : " Col_Long ",
        order : TableStore . SortOrder . SORT_ORDER _DESC
      }
    }
  ]
}

```

GeoDistanceSort

You can use GeoDistanceSort to sort the query result by geographical location. Set index sorting fields:

```

{
  sorters : [
    {
      geoDistanceSort : {
        fieldName : " Col_Geo_Po int ",
        points : [" 0 , 0 "],// Set the coordinate s
of a central point .
        order : TableStore . SortOrder . SORT_ORDER _ASC
// The query result is sorted in ascending order of
the distances between geographic al locations and the
central point .
      }
    }
  ]
}

```

```
}
```

Page scanning

Only fields with `enableSortAndAgg` set to `true` can be sorted. You can set multiple filtering conditions for index sorting.

Use Limit and Offset

When the total number of rows to be obtained is smaller than 2,000, you can set `Limit` and `Offset` to scan no more than 2,000 rows and return the result.

```
/**
 * Set Offset to 90 and Limit to 10 to scan data
 * from Line 90 to Line 100 .
 */
client . search ({
  tableName : TABLE_NAME ,
  indexName : INDEX_NAME ,
  searchQuery : {
    offset : 90 ,
    limit : 10 ,
    query : {
      queryType : TableStore . QueryType . MATCH_ALL_ QUERY
    }
  },
  getTotalCount : true // If TotalCount is set
to true , the total number of rows that meet the
filtering conditions is returned . If TotalCount is set
to false , the total number of rows that meet the
filtering conditions is not returned .
},
  columnToGet : { // Set RETURN_SPECIFIED to return a
specified number of columns , RETURN_ALL to return all
columns , or RETURN_NONE to return no data .
  returnType : TableStore . ColumnRetur nType . RETURN_ALL
},
  function ( err , data ) {
    if ( err ) {
      console . log ( ' error : ' , err );
      return ;
    }
    console . log ( ' success : ' , JSON . stringify ( data , null ,
2 ));
  });
```

Use a token

If the operation does not complete reading the data that meets the filtering conditions , the server returns `NextToken`. You can use `NextToken` to continue reading the subsequent data. The sorting method cannot be set if a token is used. When the token is used, the sorting method used is the same as the previous request, regardless of the default sorting method used by the system or the sorting method defined by the

user. Additionally, you cannot set Offset when a token is used. Data is scanned page by page, which results in a slow query.

```
/**
 * The following code provides an example of how to
 * use tokens to scan pages in sync and async modes .
 */
var params = {
  tableName : TABLE_NAME ,
  indexName : INDEX_NAME ,
  searchQuery : {
    offset : 0 ,
    limit : 10 ,
    token : null ,// Obtain a NextToken as the
starting point to scan the next page .
    query : {
      queryType : TableStore . QueryType . MATCH_ALL_ QUERY
    },
    getTotalCount : true
  },
  columnToGet : {
    returnType : TableStore . ColumnReturnType . RETURN_NON_EMPTY ,
    returnNames : [" pic_tag ", " pic_description ", "
time_stamp ", " pos "]
  }
};

/**
 * Use tokens to scan pages in the sync mode .
 */
( async () => {
  try {
    var data = await client . search ( params );
    console . log ( data );

    while ( data . nextToken ) { // If a NextToken exists ,
there is still data which is not obtained .
      params . searchQuery . token = data . nextToken ;// Update
the token value to continue scanning pages .
      data = await client . search ( params );
      console . log ( data );
    }
  } catch ( error ) {
    console . log ( error );
  }
})();

/**
 * Use tokens to scan pages in the async mode .
 */
client . search ( params , function ( err , data ) { // Update
the token value .
  console . log ( ' success :', JSON . stringify ( data , null ,
2 ));

  if ( data . nextToken ) {
    params . searchQuery . token = data . nextToken ;//
Update the token value to continue scanning pages .
    client . search ( params , function ( err , data ) {
      console . log ( ' token success :', JSON . stringify (
data , null , 2 ));
    });
  }
});
```

```

    });
  }
});

/**
 * Implement persistent storage for tokens .
 * 1 ) NextTokens are of the buffer type . Encode the
 *    NextTokens as strings based on Base64 to implement
 *    persistent storage for these tokens .
 * 2 ) Convert the string tokens to buffer and use
 *    the buffer tokens as parameters .
 */
var stringToken = data . nextToken . toString ( " base64 " ,
data . nextToken . offset , data . nextToken . limit );
var bufferToken = new Buffer ( nextToken , " base64 " );

```

5.9 Error handling

Method

The Table Store NodeJS SDK uses the `Exception` method to handle errors. Operations are successful if the called API does not throw an exception. If an exception is thrown, the operation fails.



Notice:

Batch operation APIs such as `BatchGetRow` and `BatchWriteRow` are successfully called only when the system checks that the status of each row is successful.

Exception

In the Table Store NodeJS SDK, all errors are processed in a unified manner, and are returned to the `err` parameter of the callback method. Therefore, you must check whether the `err` parameter has a value before obtaining the returned data. If an error is reported on the Table Store server, `RequestId` is returned, specifying the UUID that uniquely identifies the request. If the problem persists, save the `RequestId` and [open a ticket](#).

Retry

The system automatically retries the operation when an error occurs in the SDK. The default policy sets a maximum of 20 retry attempts and a maximum retry interval of 3000 ms. For more information about how the system retries the operation on throttling errors and read-related internal server errors, see `tablestore / lib / retry . js`.

6 . .NET SDK

6.1 Preface

Introduction

This document describes how to install and use C# SDK 3.0.0 for Table Store. This document assumes that you have already activated Alibaba Cloud Table Store and created an AccessKeyID and AccessKeySecret.

- If you have not activated Table Store or want to know about the service, visit the [Table Store product homepage](#).
- If you have not created an AccessKeyID and AccessKeySecret, log on to the [Alibaba Cloud Access Key console](#) to create an AccessKey.

Download the SDK package

- [SDK package](#)
- [GitHub](#)

For more information about version iterations, [click here](#).

Change updates and compatibility

For latest version 3.0.0

- Filters are added.
- The warnings generated during the compilation process are cleared.
- The useless dependent packages are cleared.
- The useless code is cleared.
- The template-called code is simplified.
- The invalid parameter checking is added.
- The user-configured parameters are trimmed.
- The userAgent header is added.
- The DLL file name Aliyun.dll is changed to Aliyun.TableStore.dll.
- The open source code is released to GitHub.

For SDKs V2.x.x:

- Incompatibility with partial interfaces: `Condition . IGNORE` , `Condition . EXPECT_EXIST` , and `Condition . EXPECT_NOT_EXIST` are deleted.
- The DLL file name `Aliyun . dll` is changed to `Aliyun . TableStore . dll` .

6.2 Initialization

The OTSClient is the client for Table Store. It provides callers with a series of methods for operating tables and reading/writing data from/to a single row or multiple rows.

Determine an endpoint

An endpoint is the domain of Alibaba Cloud Table Store in a region. It supports the following format.

Endpoint type	Description
Region address	Address of the region where a Table Store instance is located, for example, <code>https://instance.cn-hangzhou.ots.aliyuncs.com</code> .

Region address of Table Store

To query the endpoint where your Table Store instance is located, follow these steps:

1. Log on to the [Table Store console](#).
2. Go to the Instance Details page and locate the Instance Access URL, which is the endpoint of the instance.

Configure an AccessKey

To access the Alibaba Cloud Table Store service, you need a valid AccessKey (including an AccessKeyID and AccessKeySecret) for signature authentication. To obtain the AccessKey, follow these steps:

1. Register an [Alibaba Cloud](#) account on the Alibaba Cloud website.
2. Log on to the [AccessKey console](#) to apply for an AccessKey.

After you obtain the AccessKeyID and AccessKeySecret, use the endpoint of the Table Store to create a client and initialize an OTSClient instance:

API:

```
/// < summary >
/// OTSClient constructo r .
```

```

/// </ summary >
/// < param name =" endPoint "> Address of the Table Store
    service ( for example , ' https :// instance . cn - hangzhou .
    ots . aliyun . com : 80 ' ). It must start with ' https ://'.
    </ param >
/// < param name =" accessKeyI D "> AccessKeyI D of Table
    Store . </ param >
/// < param name =" accessKeyS ecret "> AccessKeyS ecret of
    Table Store . </ param >
/// < param name =" instanceNa me "> Name of your Table
    Store instance , which must be created on the Alibaba
    Cloud Table Store Console . </ param >
public OTSClient ( string endPoint , string accessKeyI D ,
    string accessKeyS ecret , string instanceNa me );

/// < summary >
/// Create an OTSClient instance using the OTSClientC
    onfig instance .
/// </ summary >
/// < param name =" config "> OTSClientC onfig instance </ param
    >
public OTSClient ( OTSClientC onfig config );

```

Example:

```

// Construct an OTSClientC onfig object
var config = new OTSClientC onfig ( Endpoint , AccessKeyI d
    , AccessKeyS ecret , InstanceNa me );

// Disable log output ( this function is enabled by
    default )
config . OTSDebugLo gHandler = null ;
config . OTSErrorLo gHandler = null ;

// Create an OTSClient object using OTSClientC onfig
var otsClient = new OTSClient ( config );

// Use the OTSClient to insert or query data

```



Note:

- You can set **ConnectionLimit** in the **OTSClientConfig** object. If **ConnectionLimit** is not set, the default value is 300.
- **OTSDebugLogHandler** and **OTSErrorLogHandler** in **OTSClientConfig** control the logging behavior, and are configurable.
- **RetryPolicy** in **OTSClientConfig** controls the retry logic. A default retry policy is provided. You can define your own retry policy.

Multithreading

- Multithreading is supported.
- We recommend that multiple threads use the same **OTSClient** object.

6.3 Table operations

6.3.1 CreateTable

You can call this operation to create a table based on the given table schema.

When creating a table in Table Store, you must specify the primary key for the table. A primary key contains one to four primary key columns. Each primary key column has a name and a type.

Operation

```
/// < summary >
/// Create a table based on the given table
informatio n , including the table name , primary key ,
and reserved read / write throughput .
/// < / summary >
/// < param name = " request " > Request parameter .< / param >
/// < returns > Informatio n returned by CreateTabl e .
In this example , the returned result is null and
contains no specific informatio n .
/// < / returns >
public CreateTabl eResponse CreateTabl e ( CreateTabl
eRequest request );

    /// < summary >
    /// Asynchro nous form of CreateTabl e .
    /// < / summary >
    public Task < CreateTabl eResponse > CreateTabl eAsync (
CreateTabl eRequest request );
```



Note:

After a table is created in Table Store, it takes several seconds to fully load the table. Do not perform any operations during this period.

Example

The following code provides an example of how to create a table with two primary key columns. Set the reserved read/write throughput to 0.

```
// Create a schema for the primary key columns ,
including the number , names , and types of the primary
key columns
// The first primary key column ( partition column ) is
named PK0 and belongs to the integer type .
// The second primary key column is named PK1 and
belongs to the string type .
var primaryKey Schema = new PrimaryKey Schema ();
primaryKey Schema . Add ( " pk0 " , ColumnValu eType . Integer );
primaryKey Schema . Add ( " pk1 " , ColumnValu eType . String );
```

```
// Create a tableMeta based on the table name and
the schema for the primary key columns .
var tableMeta = new TableMeta (" SampleTabl e ", primaryKey
Schema );

// Set the reserved read / write throughput to 0 .
var reservedTh  roughput = new CapacityUn it ( 0 , 0 );

try
{
    // Construct the CreateTable eRequest object .
    var request = new CreateTable eRequest ( tableMeta ,
reservedTh  roughput );

    // Call the CreateTable e operation of the OTSClient
. If no exception is thrown , the table is created .
Otherwise , the table fails to be created .
    otsClient . CreateTable e ( request );

    Console . WriteLine ( " Create    table    succeeded ." ) ;
}
// Handle exceptions .
catch ( Exception ex )
{
    Console . WriteLine ( " Create    table    failed ,    exception :{ 0
}", ex . Message );
}
```

For more information about the full sample code, see [CreateTable@GitHub](#).

6.3.2 ListTable

You can call this operation to obtain the names of all tables created in the current instance.

Operation

```
/// < summary >
/// Obtain the names of all tables created in the
current instance .
/// < / summary >
/// < param name =" request "> Request    parameter .< / param >
/// < returns > Informatio n    returned    by ListTable    and
used to obtain the table name list .< / returns > < /
returns >
public ListTableR esponse ListTable ( ListTableR equest
request );

/// < summary >
/// Asynchrono us    form    of ListTable .
/// < / summary >
public Task < ListTableR esponse > ListTableA sync (
ListTableR equest request );
```

Example

The following code provides an example of how to obtain the names of all tables in an instance:

```
var request = new ListTableRequest();
try
{
    var response = otsClient.ListTable(request);
    foreach (var tableName in response.TableNames)
    {
        Console.WriteLine("Table name:{0}", tableName);
    }
    Console.WriteLine("List table succeeded.");
}
catch (Exception ex)
{
    Console.WriteLine("List table failed, exception:{0}", ex.Message);
}
```

6.3.3 UpdateTable

You can call this operation to update the reserved read/write throughput of the specified table.

Operation

```
/// < summary >
/// Update the reserved read / write throughput of the
specified table . The new setting takes effect within
1 minute after the throughput is updated .
/// < / summary >
/// < param name =" request "> Request parameter , including
the table name and reserved read / write throughput .< /
param >
/// < returns > Contain the updated reserved read / write
throughput and other information .< / returns >
public UpdateTableResponse UpdateTable ( UpdateTable
eRequest request );

/// < summary >
/// Asynchronous form of UpdateTable .
/// < / summary >
public Task < UpdateTableResponse > UpdateTableAsync (
UpdateTable eRequest request );
```

Example

The following code provides an example of how to update the read CU and write CU of a table to 1 and 2, respectively:

```
// Update the reserved read throughput to 1 and the
reserved write throughput to 2 .
var reservedThroughput = new CapacityUnit ( 1 , 2 );

// Construct the UpdateTableRequest object .
var request = new UpdateTableRequest ( " SampleTable ",
reservedThroughput );
```

```

try
{
    // Call this operation to update the reserved read
    / write throughput of the table .
    otsClient . UpdateTable ( request );

    // If no exception is thrown , the execution is
    successful .
    Console . WriteLine ( " Update table succeeded . " );
}
catch ( Exception ex )
{
    // If task execution fails , an exception is thrown
    and an error message is displayed .
    Console . WriteLine ( " Update table failed , exception : { 0
} " , ex . Message );
}

```

For more information about the full sample code, see [UpdateTable@GitHub](#).

6.3.4 DescribeTable

You can call this operation to query the schema and reserved read/write throughput of the specified table.

Operation

```

/// < summary >
/// Query the schema and reserved read / write
throughput of the specified table .
/// < / summary >
/// < param name =" request "> Request parameter , including
the table name .< / param >
/// < returns > Contain the schema and reserved read /
write throughput of the specified table .< / returns > < /
returns >
public DescribeTableResponse DescribeTable ( DescribeTableRequest request );

/// < summary >
/// Asynchronous form of DescribeTable .
/// < / summary >
public Task < DescribeTableResponse > DescribeTableAsync (
DescribeTableRequest request );

```

Example

The following code provides an example of how to obtain the descriptive information of a table:

```

try
{
    var request = new DescribeTableRequest ( " SampleTable " );
    var response = otsClient . DescribeTable ( request );

    // Display the descriptive information of the
    specified table .
}

```

```

        Console . WriteLine ( " Describe table succeeded ." ) ;
        Console . WriteLine ( " LastIncreaseTime : { 0 }" , response .
ReservedThroughputDetails . LastIncreaseTime ) ;
        Console . WriteLine ( " LastDecreaseTime : { 0 }" , response .
ReservedThroughputDetails . LastDecreaseTime ) ;
        Console . WriteLine ( " NumberOfDecreasesToday : { 0 }" ,
response . ReservedThroughputDetails . LastIncreaseTime ) ;
        Console . WriteLine ( " ReadCapacity : { 0 }" , response .
ReservedThroughputDetails . CapacityUnit . Read ) ;
        Console . WriteLine ( " WriteCapacity : { 0 }" , response .
ReservedThroughputDetails . CapacityUnit . Write ) ;
    }
    catch ( Exception ex )
    {
        // If task execution fails , an exception is thrown
        and an error message is displayed .
        Console . WriteLine ( " Describe table failed , exception : {
0 }" , ex . Message ) ;
    }

```

For more information about the full sample code, see [DescribeTable@GitHub](#).

6.3.5 DeleteTable

You can call this operation to delete a specified table from an instance.

Operation

```

/// < summary >
/// Delete the specified table from an instance .
/// < / summary >
/// < param name = " request " > Request parameter , including
the table name . < / param >
/// < returns > Information returned by DeleteTable .
In this example , the returned result is null and
contains no specific information .
/// < / returns >
public DeleteTableResponse DeleteTable ( DeleteTable
eRequest request ) ;

/// < summary >
/// Asynchronous form of DeleteTable .
/// < / summary >
public Task < DeleteTableResponse > DeleteTableAsync (
DeleteTableeRequest request ) ;

```

Example

The following code provides an example of how to delete a table:

```

var request = new DeleteTableeRequest ( " SampleTable " ) ;
try
{
    otsClient . DeleteTable ( request ) ;
    Console . WriteLine ( " Delete table succeeded ." ) ;
}
catch ( Exception ex )
{
    Console . WriteLine ( " Delete table failed , exception : { 0
} " , ex . Message ) ;
}

```

```
}
```

For more information about the full sample code, see [DeleteTable@GitHub](#).

6.3.6 CreateSearchIndex

You can call this operation to create one or more Searchindex structures on a table. You can specify the index name and index structure when creating the Searchindex structure. The parameters for the CreateSearchIndex operation are as follows:

- **TableName**: the name of the table on which you want to create a Searchindex structure.
- **IndexName**: the name of the Searchindex structure.
- **IndexSchema**: defines the index structure and attributes of an index.
 - **IndexSetting**: includes the RoutingFields parameter. You can use this advanced optional feature to customize a routing. You can specify some primary keys as routing fields. Table Store distributes data that is written to an index to different partitions according to the specified routing fields. The data with the same routing field value is distributed to the same partition.
 - **FieldSchemas**: the FieldSchema list. You can specify the following parameters for each FieldSchema.
 - **FieldName** (required): a string that indicates the name of a field for indexing. The field can be a primary key column or an attribute column.
 - **FieldType** (required): the field type. For more information about this parameter, see API reference of Searchindex.
 - **Index** (optional): a Boolean value that indicates whether to enable indexing on the target field.
 - **IndexOptions** (optional): the index setting.
 - **Analyzer** (optional): the tokenizer setting.
 - **EnableSortAndAgg** (optional): a Boolean value that indicates whether to enable sorting and statistics.
 - **Store** (optional): a Boolean value that indicates whether to store the value of the target field to a Searchindex structure. If you set this parameter to true, you can read data directly from the index without querying the primary table. This optimizes query performance.
 - **Array** (optional): a Boolean value that indicates whether the column is an array. A value of true indicates that the column is an array. Data written to

the column must be a JSON array, such as ["a", "b", "c"]. A NESTED column is an array, and does not require the Array parameter.

- **IndexSort (optional):** indicates the presorting method for indexed data. You can sort data by primary key or column. By default, you sort data by primary key.

The sample code is as follows:

```

/// < summary >
/// Create a Searchindex structure that consists of
the Keyword_type_col , Long_type_col , and Text_type_col
columns . Then , set the data type of these columns
to KEYWORD , LONG , and TEXT , respective ly .
/// </ summary >
/// < param name =" otsClient "></ param >
public static void CreateSearchIndex ( OTSClient otsClient
)
{
    // Set the table name and index name .
    CreateSearchIndexRequest request = new CreateSearch
chIndexRequest ( TableName , IndexName );
    List < FieldSchema > FieldSchemas = new List <
FieldSchema >() {
        new FieldSchema ( Keyword_type_col , FieldType .
KEYWORD ){ // Set the field name and field type .
            index = true , // Set the parameter to true
to enable indexing .
            EnableSortAndAgg = true // Set the parameter
to true to enable sorting and statistics .
        },
        new FieldSchema ( Long_type_col , FieldType . LONG ){
index = true , EnableSortAndAgg = true },
        new FieldSchema ( Text_type_col , FieldType . TEXT ){
index = true }
    };
    request . IndexSchema = new IndexSchema ()
    {
        FieldSchemas = FieldSchemas
    };
    // Call the OTSClient to create a Searchindex
structure .
    CreateSearchIndexResponse response = otsClient .
CreateSearchIndex ( request );
    Console . WriteLine ( " Searchindex is created : " +
IndexName );
}

```

The following code provides an example of how to specify the IndexSort parameter:

```

/// < summary >
/// Create a Searchindex structure that consists of
the Keyword_type_col , Long_type_col , and Text_type_col
columns . Then , set the data type of these columns
to KEYWORD , LONG , and TEXT , respective ly .
/// </ summary >
/// < param name =" otsClient "></ param >
public static void CreateSearchIndexWithIndexSort (
OTSClient otsClient )
{

```

```

// Set the table name and index name .
CreateSearchIndexRequest request = new CreateSearchIndexRequest ( TableName , IndexName );
List< FieldSchema > FieldSchemas = new List<
FieldSchema >() {
    new FieldSchema ( Keyword_type_col , FieldType .
KEYWORD ){ // Set the field name and field type .
        index = true , // Set the parameter to true
to enable indexing .
        EnableSort AndAgg = true // Set the parameter
to true to enable sorting and statistics .
    },
    new FieldSchema ( Long_type_col , FieldType . LONG ){
index = true , EnableSort AndAgg = true },
    new FieldSchema ( Text_type_col , FieldType . TEXT ){
index = true }
};
request . IndexSchema = new IndexSchema ()
{
    FieldSchemas = FieldSchemas ,
    // Presort data by the Long_type_col column .
You must index the Long_type_col column and enable
sorting and statistics .
    IndexSort = new DataModel . Search . Sort . Sort ()
    {
        Sorters = new List< DataModel . Search . Sort .
ISorter >
        {
            new DataModel . Search . Sort . FieldSort (
Long_type_col , DataModel . Search . Sort . SortOrder . ASC )
        }
    };
};

CreateSearchIndexResponse response = otsClient .
CreateSearchIndex ( request );
Console . WriteLine ( " Searchindex is created : " +
IndexName );
}

```

6.3.7 DescribeSearchIndex

DescribeSearchIndex

You can call this operation to obtain the details of the Searchindex structure of a table , such as the description of attributes and index configurations. The sample code is as follows:

```

/// < summary >
/// Obtain the details of the Searchindex structure .
/// < / summary >
/// < param name =" otsClient ">< / param >
public static void DescribeSearchIndex ( OTSClient
otsClient )
{
    // Set the table name and index name .
    DescribeSearchIndexRequest request = new DescribeSearchIndexRequest ( TableName , IndexName );

```

```

        DescribeSearchIndexResponse response = otsClient .
        DescribeSearchIndex ( request );
        string serializedObjectString = JsonConvert .
        SerializeObject ( response );
        Console . WriteLine ( serializedObjectString ); // Display
        the details of the response .
    }

```

6.3.8 ListSearchIndex

ListSearchIndex

You can call this operation to retrieve the list of all Searchindex structures created for a table. The sample code is as follows:

```

/// < summary >
/// List Searchindex structures .
/// </ summary >
/// < param name =" otsClient "></ param >
public static void ListSearchIndex ( OTSClient otsClient )
{
    // Set the name of the table .
    ListSearchIndexRequest request = new ListSearch
    IndexRequest ( TableName );
    ListSearchIndexResponse response = otsClient .
    ListSearchIndex ( request );
    // Table Store returns all Searchindex structures
    created for the specified table .
    foreach ( var index in response . IndexInfos )
    {
        Console . WriteLine ( " indexname :" + index . IndexName );
        Console . WriteLine ( " tablename :" + index . TableName );
    }
}

```

6.3.9 DeleteSearchIndex

DeleteSearchIndex

You can call this operation to delete a Searchindex structure. The sample code is as follows:

```

/// < summary >
/// Delete a Searchindex structure .
/// </ summary >
/// < param name =" otsClient "></ param >
public static void DeleteSearchIndex ( OTSClient otsClient )
{
    // Set the table name and index name .
    DeleteSearchIndexRequest request = new DeleteSearch
    IndexRequest ( TableName , IndexName );
    // Call the OTSClient to delete the target
    Searchindex structure .
    DeleteSearchIndexResponse response = otsClient .
    DeleteSearchIndex ( request );
}

```

```
}
```

6.4 Searchindex operations

6.4.1 TermQuery

TermQuery

You can call this operation to query data that exactly matches the specified value of an attribute. When you query a TEXT string, Table Store tokenizes the string and exactly matches any of the tokens. For example, Table Store tokenizes a TEXT string "tablestore is cool" into "tablestore", "is", and "cool". When you specify any of these tokens as a query string, you can retrieve query results that contain "tablestore is cool".

```
/// < summary >
/// Search the Keyword_ty pe_col column for data that
/// exactly matches the query string SearchInde x .
/// < / summary >
/// < param name =" otsClient ">< / param >
public static void TermQuery ( OTSClient otsClient )
{
    var searchQuer y = new SearchQuer y ();
    // Table Store returns the total count of query
    results .
    searchQuer y . GetTotalCo unt = true ;
    // Set the query type to TermQuery , the query
    range to Keyword_ty pe_col , and the query string to
    SearchInde x .
    searchQuer y . Query = new TermQuery ( Keyword_ty pe_col ,
    new ColumnValu e (" SearchInde x "));

    var request = new SearchRequ est ( TableName , IndexName
    , searchQuer y );
    request . ColumnsToG et = new ColumnsToG et ()
    {
        ReturnAll = true // Table Store returns all
        columns of query results .
    };
    var response = otsClient . Search ( request );
    // Table Store returns the NextToken string to
    check whether there are additional available results .
```

6.4.2 Query by match

MatchAllQuery

You can call this operation to check the total number of rows in a table or check some rows in the table.

```
/// < summary >
```

```

/// Search all rows to obtain the number of rows
/// that meet the conditions .
/// </ summary >
/// < param name =" otsClient "></ param >
public static void MatchAllQuery ( OTSClient otsClient )
{
    var searchQuery = new SearchQuery ();
    searchQuery . Query = new MatchAllQuery ();
    searchQuery . GetTotalCount = true ; // Set the
    GetTotalCount parameter to true to obtain the number
    of rows that meet the conditions .
    /*
    * In a MatchAllQuery result set , the value of
    TotalCount is the total number of rows in a table
    . This value is an approximate value when you
    query a table with a large number of rows .
    * If you only want to query TotalHit , set Limit
    to 0 to specify that no row of data is returned .
    */
    searchQuery . Limit = 0 ;
    var request = new SearchRequest ( TableName , IndexName
    , searchQuery );

    var response = otsClient . Search ( request );
    // Check whether Table Store returns matched data
    from all partitions . When the value of isAllSuccess
    is false , Table Store may fail to query some
    partitions and return only a part of data .
    Console . WriteLine ( " IsAllSuccess : " + response .
    IsAllSuccess );
    Console . WriteLine ( " Total Count : " + response . TotalCount
    );
}

```

MatchQuery

You can call this operation to query a table based on approximate matches. For example, you specify a query string "this is". Table Store returns query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".

```

/// < summary >
/// Create a fuzzy match based on a phrase or
/// adjacent query . Table Store returns query results in
/// a specified column .
/// </ summary >
/// < param name =" otsClient "></ param >
public static void MatchQuery ( OTSClient otsClient )
{
    var searchQuery = new SearchQuery ();
    // Set the query type to MatchQuery , the query
    range to Text_type_col , and the query string to
    SearchIndex .
    searchQuery . Query = new MatchQuery ( Text_type_col , "
    SearchIndex " );
    searchQuery . GetTotalCount = true ;
    var request = new SearchRequest ( TableName , IndexName
    , searchQuery );
    request . ColumnsToGet = new ColumnsToGet ()
    {

```

```

        Columns = new List<string>() { Long_type_col ,
Text_type_col , Keyword_type_col }
    };

    var response = otsClient . Search ( request );

    Console . WriteLine ( " Total Count : " + response . TotalCount
    );
}

```

MatchPhraseQuery

You can call this operation to query data that matches a phrase. **MatchPhraseQuery** is similar to **MatchQuery**, only that Table Store matches a phrase as an entity for **MatchPhraseQuery**. For example, you specify a query string "this is". Table Store returns query results such as "..., this is tablestore" and "this is a table", but does not return query results such as "this table is..." and "is this a table".

```

/// < summary >
/// MatchPhras eQuery is similar to MatchQuery , only
/// that Table Store matches words in a phrase for
/// MatchQuery , but matches a phrase as an entity for
/// MatchPhras eQuery .
/// < / summary >
/// < param name =" otsClient ">< / param >
public static void MatchPhras eQuery ( OTSClient otsClient )
{
    var searchQuer y = new SearchQuer y ();
    // Set the query type to MatchPhras eQuery .
    searchQuer y . Query = new MatchPhras eQuery ( Text_type_
col , " TableStore SearchInde x " );
    searchQuer y . GetTotalCo unt = true ;
    var request = new SearchRequ est ( TableName , IndexName
, searchQuer y );
    request . ColumnsToG et = new ColumnsToG et ()
    {
        Columns = new List<string>() { Long_type_col ,
Text_type_col , Keyword_ty pe_col }
    };

    var response = otsClient . Search ( request );

    Console . WriteLine ( " Total Count : " + response . TotalCount
    );
}

```

6.4.3 PrefixQuery

You can call this operation to query data that matches a prefix that you specify in a query. When a table contains a TEXT string, Table Store tokenizes the string and matches any of the tokens with the specified prefix.

```

/// < summary >
/// Create a PrefixQuer y structure .
/// < / summary >
/// < param name =" otsClient ">< / param >

```

```

public static void PrefixQuery ( OTSClient otsClient )
{
    var searchQuery = new SearchQuery ();
    // Set the query type to PrefixQuery, the query
    range to Keyword_type_col, and the prefix to Search .
    searchQuery . Query = new PrefixQuery ( Keyword_type_col , " Search " );
    searchQuery . GetTotalCount = true ;
    var request = new SearchRequest ( TableName , IndexName , searchQuery );
    request . ColumnsToGet = new ColumnsToGet ()
    {
        ReturnAll = true // Table Store returns all
        columns of query results .
    };

    var response = otsClient . Search ( request );

    Console . WriteLine ( " Total Count : " + response . TotalCount );
}

```

6.4.4 RangeQuery

RangeQuery

You can call this operation to query data that falls within a range that you specify in a query. When a table contains a TEXT string, Table Store tokenizes the string and matches any of the tokens that falls within the specified range.

```

/// < summary >
/// Create a RangeQuery structure . Specify a range .
/// Search for data that falls within the range .
/// < / summary >
/// < param name = " otsClient ">< / param >
public static void RangeQuery ( OTSClient otsClient )
{
    var searchQuery = new SearchQuery ();
    searchQuery . GetTotalCount = true ;
    var rangeQuery = new RangeQuery ( Long_type_col , new
    ColumnValue ( 0 ), new ColumnValue ( 6 ));
    // The value includes the lower boundary value ,
    which is greater than or equal to 0 .
    rangeQuery . IncludeLower = true ;
    searchQuery . Query = rangeQuery ;
    var request = new SearchRequest ( TableName , IndexName , searchQuery );
    var response = otsClient . Search ( request );

    Console . WriteLine ( " Total Count : " + response . TotalCount );
}

```

```
}
```

6.4.5 WildcardQuery

WildcardQuery

You can call this operation to match wildcard characters in a query. You can specify a value to be matched as a string with one or more wildcard characters. An asterisk (*) represents any length of characters. A question mark (?) represents any single character. For example, when you query a string "table*e", Table Store returns query results such as "tablestore". The specified string for matching cannot start with an asterisk (*).

```
/// < summary >
/// Create a WildcardQuery structure . An asterisk (*)
/// represents any length of characters . A question mark
/// (?) represents any single character .
/// < / summary >
/// < param name = " otsClient ">< / param >
public static void WildcardQuery ( OTSClient otsClient )
{
    var searchQuery = new SearchQuery ();
    // Set the query type to WildcardQuery . The query
    // value can contain wildcards .
    searchQuery . Query = new WildcardQuery ( Keyword_type_col , "* Search *");
    searchQuery . GetTotalCount = true ;
    var request = new SearchRequest ( TableName , IndexName , searchQuery );
    request . ColumnsToGet = new ColumnsToGet ()
    {
        ReturnAll = true
    };
    var response = otsClient . Search ( request );
    Console . WriteLine ( " Total Count : " + response . TotalCount );
}
```

6.4.6 Query by geographic location

A Searchindex structure supports three geographic location-based query methods: GeoBoundingBoxQuery, GeoDistanceQuery, and GeoPolygonQuery.

GeoBoundingBoxQuery

You can call this operation to query data that falls within a geographical rectangle that you specify in a query. Table Store returns the rows where the value of an attribute falls within a geographical rectangle.

```
/// < summary >
```

```

/// Data in the Geo_type_c ol column belongs to the
/// GeoPoint type . Specify a geographic al rectangle . Set
/// coordinate s of its top - left vertex to " 10 , 0 "
/// and coordinate s of its bottom - right vertex to " 0 ,
/// 10 ". Search the Geo_type_c ol column for data that
/// falls within the geographic al rectangle area .
/// </ summary >
/// < param name =" client "></ param >
public static void GeoBoundin gBoxQuery ( OTSClient client )
{
    SearchQuer y searchQuer y = new SearchQuer y ();
    GeoBoundin gBoxQuery geoBoundin gBoxQuery = new
    GeoBoundin gBoxQuery (); // Set the query type to
    GeoBoundin gBoxQuery .
    geoBoundin gBoxQuery . FieldName = Geo_type_c ol ; //
    Specify the field for indexing .
    geoBoundin gBoxQuery . setTopLeft ( " 10 , 0 " ); // Specify
    coordinate s for the top - left vertex of the
    rectangle .
    geoBoundin gBoxQuery . setBottomR ight ( " 0 , 10 " ); //
    Specify coordinate s for the bottom - right vertex of
    the rectangle .
    searchQuer y . Query = geoBoundin gBoxQuery ;

    SearchRequ est searchRequ est = new SearchRequ est (
    TableName , IndexName , searchQuer y );

    var columnsToG et = new ColumnsToG et ();
    columnsToG et . Columns = new List < string > { Geo_type_c
    ol }; // Specify the Col_GeoPoi nt column as the
    column where Table Store returns query results .
    searchRequ est . ColumnsToG et = columnsToG et ;

    SearchResp onse response = client . Search ( searchRequ est
    );
    Console . WriteLine ( response . TotalCount );
}

```

GeoDistanceQuery

You can call this operation to query data that falls within a specified distance from a specified central point. Table Store returns the rows where the value of an attribute falls within the distance from the central point.

```

/// < summary >
/// Search the Geo_type_c ol column for data that
/// falls within the specified distance from the specified
/// central point .
/// </ summary >
/// < param name =" client "></ param >
public static void GeoDistanc eQuery ( OTSClient client )
{
    SearchQuer y searchQuer y = new SearchQuer y ();
    GeoDistanc eQuery geoDistanc eQuery = new GeoDistanc
    eQuery (); // Set the query type to GeoDistanc eQuery .
    geoDistanc eQuery . FieldName = Geo_type_c ol ;
    geoDistanc eQuery . CenterPoin t = " 10 , 11 "; // Specify
    coordinate s for a central point .
    geoDistanc eQuery . setDistanc eInMeter ( 10000 ); // You
    can specify 10 , 000 meters as the farthest distance
    from the central point .
}

```

```

        searchQuery.Query = geoDistanceQuery;

        SearchRequest searchRequest = new SearchRequest (
            TableName, IndexName, searchQuery);

        ColumnsToGet columnsToGet = new ColumnsToGet ();
        columnsToGet.Columns = new List<string>() {
            Geo_type_column }; // Specify the Col_GeoPoint column as
            the column where Table Store returns query results .
        searchRequest.ColumnsToGet = columnsToGet;

        SearchResponse response = client.Search ( searchRequest
    );
    Console.WriteLine ( response.TotalCount );
}

```

GeoPolygonQuery

You can call this operation to query data that falls within a polygon area that you specify in a query. Table Store returns the rows where the value of an attribute falls within a polygon area.

```

/// < summary >
/// Search the Geo_type_column column for data that
/// falls in a specified polygon area .
/// </ summary >
/// < param name =" client "></ param >
public static void GeoPolygon Query ( OTSClient client )
{
    SearchQuery searchQuery = new SearchQuery ();
    GeoPolygon Query geoPolygon Query = new GeoPolygon Query
    (); // Set the query type to GeoPolygon Query .
    geoPolygon Query.FieldName = Geo_type_column ;
    geoPolygon Query.Points = new List<string>() { " 0 ,
    0 ", " 10 , 0 ", " 10 , 10 " }; // Specify coordinate s for
    vertices of a polygon .
    searchQuery.Query = geoPolygon Query ;

    SearchRequest searchRequest = new SearchRequest (
        TableName, IndexName, searchQuery);

    ColumnsToGet columnsToGet = new ColumnsToGet ();
    columnsToGet.Columns = new List<string>() {
        Geo_type_column }; // Specify the Col_GeoPoint column as
        the column where Table Store returns query results .
    searchRequest.ColumnsToGet = columnsToGet;

    SearchResponse response = client.Search ( searchRequest
    );
    Console.WriteLine ( response.TotalCount );
}

```

6.4.7 BoolQuery

You can call this operation to query data based on multiple conditions. BoolQuery may include one or more subquery conditions. You can retrieve the rows that match the subquery conditions.

You can combine these subquery conditions in different ways. If you specify these subquery conditions as MustQueries, Table Store returns results that meet all of the subquery conditions. If you specify these subquery conditions as MustNotQueries, Table Store returns results that meet none of the subquery conditions. BoolQuery has the following parameters:

BoolQuery has the following parameters:

```
/// < summary >
/// The document must match all subquery conditions .
/// </ summary >
public List < IQuery > MustQueries { get ; set ; }
/// < summary >
/// The document must match none of the subquery
/// conditions .
/// </ summary >
public List < IQuery > MustNotQueries { get ; set ; }
/// < summary >
/// The document must match all subfilters . Unlike
/// MustQueries , the matching score is ignored for
/// FilterQueries .
/// </ summary >
public List < IQuery > FilterQueries { get ; set ; }
/// < summary >
/// The document must match at least one of the
/// subquery conditions . The score is higher if the
/// document matches more subquery conditions .
/// </ summary >
public List < IQuery > ShouldQueries { get ; set ; }
/// < summary >
/// Specify the minimum number of ShouldQueries clauses
/// that the document must match . The default value is
/// 1 .
/// </ summary >
public int? MinimumShouldMatch { get ; set ; }
```

6.4.8 NestedQuery

NestedQuery

You can call this operation to query nested objects by specifying the path for the nested attribute and a subquery (which can be any query).

Example

```

/// < summary >
/// A sub - document contains two nested columns , namely
/// , nested_1 and nested_2 . Search coll_neste d . nested_1
/// for the data with the value of tablestore .
/// </ summary >
/// < param name =" otsClient "></ param >
public static void NestedQuer y ( OTSClient otsClient )
{
    var searchQuer y = new SearchQuer y ();
    searchQuer y . GetTotalCo unt = true ;
    var nestedQuer y = new NestedQuer y ();
    nestedQuer y . Path = " coll_neste d "; // Set the path
    for the nested attribute .
    TermQuery termQuery = new TermQuery ( " coll_neste d .
    nested_1 " , new ColumnValu e ( " tablestore " )); // Create a
    subquery for the NestedQuer y structure .
    nestedQuer y . Query = termQuery ;
    nestedQuer y . ScoreMode = ScoreMode . None ;

    var request = new SearchRequ est ( TableName , IndexName
, searchQuer y );
    request . ColumnsToG et = new ColumnsToG et ()
    {
        ReturnAll = true
    };

    var response = otsClient . Search ( request );

    Console . WriteLine ( " Total Count : " + response . TotalCount
);
}

```

6.4.9 Sorting and paging

indexSort

The `indexSort` parameter indicates the presorting method. After you create a `Searchindex` structure on a table, Table Store presorts data according to the `indexSort` parameter by default. (A `Searchindex` structure with nested fields does not support the `indexSort` parameter and presorting.) Table Store returns query results by primary key by default. You can also set the `indexSort` parameter to specify the sorting method when creating a `Searchindex` structure. This parameter determines the default sequence in which Table Store returns query results. Table Store returns results by primary key by default.

Sorting methods

You can specify the sorting method for each query. The `Searchindex` structure supports the following four sorting methods. By specifying multiple sorting methods , you can sort data by one method and then by another.

ScoreSort

Sort data by score. This method applies to correlation-based indexing such as full-text indexing. You must specify the `ScoreSort` parameter so that you can sort data by score of correlation. Otherwise, query results are returned in the sequence indicated by the `indexSort` parameter.

```
var searchQuery = new SearchQuery();
searchQuery.Sort = new Sort(new List<ISorter>() { new
    ScoreSort() });
```

PrimaryKeySort

Sort data by primary key.

```
Ascending :
var searchQuery = new SearchQuery();
searchQuery.Sort = new Sort(new List<ISorter>() { new
    PrimaryKeySort() });

Descending :
var searchQuery = new SearchQuery();
searchQuery.Sort = new Sort(new List<ISorter>() { new
    PrimaryKeySort(SortOrder.DESC) });
```

FieldSort

Sort data by a column.

```
var searchQuery = new SearchQuery();
var fieldSort = new FieldSort("col", SortOrder.ASC);
searchQuery.Sort = new Sort(new List<ISorter>() {
    fieldSort });
```

Sort data by a column and then by another.

```
var searchQuery = new SearchQuery();
var col1Sort = new FieldSort("col", SortOrder.ASC);
var col2Sort = new FieldSort("col2", SortOrder.ASC);
searchQuery.Sort = new Sort(new List<ISorter>() {
    col1Sort, col2Sort });
```

GeoDistanceSort

Sort data by geographical distance.

```
var searchQuery = new SearchQuery();
var geoDistanceSort = new GeoDistanceSort("geoCol", new
    List<string>() {"0", "0"});
```

```
searchQuery.Sort = new Sort ( new List < ISorter >() {
    geoDistanceSort });
```

Paging methods

Set Limit and Offset

When the total number of rows to be obtained is smaller than or equal to 2,000, we recommend you set the Limit and Offset parameters for paging. (The total sum of the values of the Limit and Offset parameters must be smaller than or equal to 2,000.)

```
var searchQuery = new SearchQuery ();
searchQuery.Query = new MatchAllQuery ();
searchQuery.Limit = 100 ;
searchQuery.Offset = 100 ;
```

Use a token

If additional data records are available apart from returned results, Table Store will return a NextToken string. You can use the NextToken string to receive query results on the next page. After you use a token, the sorting method is the same as that of the last query. (The sorting method of the last query may be indicated by the default value of the indexSort parameter or specified by you.) In this case, you cannot specify the sorting method. In addition, you cannot set the Offset parameter after using a token. You can only retrieve data page by page without skipping.

```
/// < summary >
/// In this example , a token is used for paging .
/// Table Store returns all data records into a list .
/// </ summary >
/// < param name =" otsClient "></ param >
public static SearchResponse ReadMoreRowsWithToken (
    OTSClient otsClient )
{
    var searchQuery = new SearchQuery ();
    searchQuery.Query = new MatchAllQuery ();

    var request = new SearchRequest ( TableName , IndexName
    , searchQuery );

    var response = otsClient . Search ( request );
    var rows = response . Rows ;
    while ( response . NextToken != null ) // The value of
    the NextToken string is null , indicating that all
    data records are retrieved .
    {
        request . SearchQuery . Token = response . NextToken ;
        response = otsClient . Search ( request );
        rows . AddRange ( response . Rows );
    }

    return response ;
}
```

```
}
```

6.5 Single-row operations

The Table Store SDK provides the following single-row operation interfaces: PutRow, GetRow, UpdateRow, and DeleteRow.

PutRow

Inserts data into the specified row.

API

```

    /// < summary >
    /// Write a data row based on the specified
    table name , primary keys , and attributes . CapacityUn it
    consumed by the operation is returned .
    /// < / summary >
    /// < param name =" request "> Data insertion request < /
    param >
    /// < returns > CapacityUn it consumed by the
    operation < / returns >
    public PutRowResp onse PutRow ( PutRowRequ est
    request );

    /// < summary >
    /// Asynchrono us form of PutRow
    /// < / summary >
    public Task < PutRowResp onse > PutRowAsyn c (
    PutRowRequ est request );

```

Example 1

Insert a data row.

```

    // Define the primary keys of the row , which
    must be consistent with the primary keys defined in
    TableMeta during table creation
    var primaryKey = new PrimaryKey ();
    primaryKey . Add ( " pk0 ", new ColumnValu e ( 0 ));
    primaryKey . Add ( " pk1 ", new ColumnValu e ( " abc "));

    // Define the attribute columns of the 100
    rows
    var attribute = new AttributeC olumns ();
    attribute . Add ( " col0 ", new ColumnValu e ( 0 ));
    attribute . Add ( " col1 ", new ColumnValu e ( " a "));
    attribute . Add ( " col2 ", new ColumnValu e ( true ));

    try
    {
        // Construct the request object for data
        insertion , with RowExisten ceExpectat ion . IGNORE indicating
        that data is still inserted even when the specified
        row does not exist
        var request = new PutRowRequ est ( " SampleTabl e
        ", new Condition ( RowExisten ceExpectat ion . IGNORE ),
        primaryKey , attribute );
    }

```

```

        // Call PutRow to insert data
        otsClient . PutRow ( request );

        // Execution is successful if no exception is
        thrown .
        Console . WriteLine ( " Put row succeeded ." );
    }
    catch ( Exception ex )
    {
        // Execution fails if an exception is thrown
        , and an error message is printed .
        Console . WriteLine ( " Put row failed , exception : {
0 } " , ex . Message );
    }

```



Note:

- Condition.IGNORE, Condition.EXPECT_EXIST, and Condition.EXPECT_NOT_EXIST are deprecated starting from v3.0.0. Use new Condition (RowExistenceExpectation.IGNORE), new Condition (RowExistenceExpectation.EXPECT_EXIST), and new Condition (RowExistenceExpectation.EXPECT_NOT_EXIST) instead.
- RowExistenceExpectation.IGNORE indicates that new data is still inserted even when the specified row does not exist. If the inserted data is the same as the existing data, the existing data is overwritten.
- RowExistenceExpectation.EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten.
- RowExistenceExpectation.EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist.
- In addition to row conditions, the Condition parameter also supports column conditions starting from v2.2.0.

Code details could be obtained at [PutRow@GitHub](#).

Example 2

Insert a data row based on specified conditions.

The following code example inserts data only when the specified row exists and the value of col1 is greater than 24.

```

// Define the primary keys of the row , which
must be consistent with the primary keys defined in
TableMeta during table creation
var primaryKey = new PrimaryKey ();
primaryKey . Add ( " pk0 " , new ColumnValue ( 0 ));
primaryKey . Add ( " pk1 " , new ColumnValue ( " abc " ));

```

```

// Define the attribute columns of the 100
rows
AttributeColumns attribute = new AttributeColumns
();
attribute.Add("col0", new ColumnValue(0));
attribute.Add("col1", new ColumnValue("a"));
attribute.Add("col2", new ColumnValue(true));

var request = new PutRowRequest(tableName, new
Condition(RowExistenceExpectation.EXPECT_EXIST),
primaryKey, attribute);

// When the value of col0 is greater than 24
, PutRow is executed again to overwrite the original
value.
try
{
    request.Condition.ColumnCondition = new
RelationalCondition("col0",
, RelationalCondition
.CompareOperator.GREATER_THAN,
new ColumnValue(
24));
    otsClient.PutRow(request);

    Console.WriteLine("Put row succeeded.");
}
catch (Exception ex)
{
    Console.WriteLine("Put row failed. error:{0
}", ex.Message);
}

```

**Note:**

- You can set a single condition or a combination of conditions. For example, you can set two conditions for data insertion as `col1 > 5` and `pk2 < 'xyz'`.
- Attribute columns and primary key columns support the Condition parameter.
- When the column specified in a condition does not exist in a row, `PassIfMissing` in `RelationCondition` controls the action to be taken. The default value is true.

Code details could be obtained at [ConditionPutRow@GitHub](#).

Example 3

Insert a data row asynchronously.

```

try
{
    var putRowTaskList = new List<Task>
PutRowResponse>();
    for (int i = 0; i < 100; i++)
    {
        // Define the primary keys of the row
, which must be consistent with the primary keys
defined in TableMeta during table creation
        var primaryKey = new PrimaryKey();
    }
}

```

```

    primaryKey . Add ( " pk0 ", new ColumnValue ( i
));
    primaryKey . Add ( " pk1 ", new ColumnValue ( "
abc "));

    // Define the attribute columns of the
100 rows
    var attribute = new AttributeColumns ();
    attribute . Add ( " col0 ", new ColumnValue ( i
));
    attribute . Add ( " col1 ", new ColumnValue ( " a
" ));
    attribute . Add ( " col2 ", new ColumnValue (
true ));

    var request = new PutRowRequest ( TableName
, new Condition ( RowExistenceExpectation . IGNORE ),
    primaryKey ,
    attribute );

    putRowTaskList . Add ( TableStoreClient .
PutRowAsync ( request ));
}

// Wait until each asynchronous call returns
results and print the consumed CU
foreach ( var task in putRowTaskList )
{
    task . Wait ();
    Console . WriteLine ( " consumed read :{ 0 }, write
:{ 1 }", task . Result . ConsumedCapacityUnit . Read ,
    task . Result . ConsumedCapacityUnit . Write );
}

// Data is inserted successfully if no
exception is thrown .
Console . WriteLine ( " Put row async succeeded .");
}
catch ( Exception ex )
{
    // An error message is printed if an
exception is thrown .
    Console . WriteLine ( " Put row async failed .
exception :{ 0 }", ex . Message );
}

```

**Notice:**

Each asynchronous call starts a thread. A timeout error may occur if too many asynchronous calls are started consecutively and each call consumes an extended period of time.

Code details can be obtained at [PutRowAsync@GitHub](#).

GetRow

Reads a single data row based on a given primary key.

API

```

    /// < summary >
    /// Read a single data row based on a given
    primary key .
    /// </ summary >
    /// < param name =" request "> Data query request </
    param >
    /// < returns > Response returned by GetRow </ returns >
    public GetRowResponse GetRow ( GetRowRequest
    request );

    /// < summary >
    /// Asynchronous form of GetRow
    /// </ summary >
    public Task < GetRowResponse > GetRowAsync (
    GetRowRequest request );

```

Example 1

Read a data row.

```

    // Define the primary keys of the row , which
    must be consistent with the primary keys defined in
    TableMeta during table creation
    PrimaryKey primaryKey = new PrimaryKey ();
    primaryKey . Add ( " pk0 " , new ColumnValue ( 0 ));
    primaryKey . Add ( " pk1 " , new ColumnValue ( " abc " ));

    try
    {
        // Construct a query request object . The
        entire row is read if no column is specified .
        var request = new GetRowRequest ( TableName ,
        primaryKey );

        // Call the GetRow interface to query data
        var response = otsClient . GetRow ( request );

        // Output the data of the row . The output
        data is omitted here . For more information , see the
        GitHub link .

        // Data is read successfully if no
        exception is thrown .
        Console . WriteLine ( " Get row succeeded . " );
    }
    catch ( Exception ex )
    {
        // Execution fails if an exception is thrown
        , and an error message is printed .
        Console . WriteLine ( " Update table failed ,
        exception :{ 0 }" , ex . Message );
    }

```

**Note:**

- If you query a data row, the system returns the data in all columns of the row. You can use the `columnsToGet` parameter to read the data in specified columns. For example, the system only returns the data in `col0` and `col1` if `col0` and `col1` are inserted into `columnsToGet`.
- Conditional filter is supported. For example, you can configure the system to return results only when the value of `col0` is greater than 24.
- When the `columnsToGet` and `Condition` parameters are both used, the system returns results first based on `columnsToGet` and then filters the returned columns based on the `Condition` parameter.
- When a specified column does not exist, `PassIfMissing` determines the next action.

Code details could be obtained at [GetRow@GitHub](#).

Example 2

Read a data row using the filter feature.

The following code example queries data and configures the system to return only the data in `col0` and `col1` and filter the data in `col0` based on the condition that the value of `col0` is 24.

```
// Define the primary keys of the row, which
// must be consistent with the primary keys defined in
// TableMeta during table creation
PrimaryKey primaryKey = new PrimaryKey ();
primaryKey . Add ( " pk0 ", new ColumnValue ( 0 ));
primaryKey . Add ( " pk1 ", new ColumnValue ( " abc "));

var rowQueryCriteria = new SingleRowQueryCriteria
( " SampleTable ");
rowQueryCriteria . RowPrimaryKey = primaryKey ;

// Condition 1 : The value of col0 is 5 .
var filter1 = new RelationalCondition ( " col0 ",
    RelationalCondition . CompareOperator . EQUAL
,
    new ColumnValue ( 5 ));

// Condition 2 : The value of col1 is not ff .
var filter2 = new RelationalCondition ( " col1 ",
    RelationalCondition . CompareOperator . NOT_EQUAL , new
    ColumnValue ( " ff "));

// Construct a combination of Condition 1 and
// Condition 2 with the OR relationship
var filter = new CompositeCondition ( CompositeC
ondition . LogicOperator . OR );
filter . AddCondition ( filter1 );
filter . AddCondition ( filter2 );
```

```

        rowQueryCriteria.Filter = filter;

        // Set the row you want to query based on
        the condition [ col0 , col1 ], and filter the queried
        data based on the specified condition
        rowQueryCriteria.AddColumns ToGet (" col0 ");
        rowQueryCriteria.AddColumns ToGet (" col1 ");

        // Construct GetRowRequest
        var request = new GetRowRequest ( rowQueryCriteria
    );

    try
    {
        // Query data
        var response = otsClient.GetRow ( request );

        // Output data or perform the related logical
        operation ( omitted )

        // Execution is successful if no exception is
        thrown .
        Console.WriteLine (" Get row with filter
        succeeded .");
    }
    catch ( Exception ex )
    {
        // Execution fails if an exception is thrown
        , and an error message is printed .
        Console.WriteLine (" Get row with filter failed
        , exception :{ 0 }", ex . Message );
    }

```

Code details could be obtained at [GetRowWithFilter@GitHub](#).

UpdateRow

Updates the data of the specified row. If the row does not exist, a new row is added. If the row exists, the values of the specified columns are added, modified, or deleted based on the request content.

API

```

    /// < summary >
    /// Update the data of the specified row . If
    the row does not exist , a new row is added . If
    the row exists , the values of the specified columns
    are added , modified , or deleted based on the request
    content .
    /// < / summary >
    /// < param name =" request "> Request instance < / param >
    public UpdateRowResponse UpdateRow ( UpdateRowRequest
    request );

    /// < summary >
    /// Asynchronous form of UpdateRow .
    /// < / summary >
    /// < param name =" request ">< / param >
    /// < returns >< / returns >

```

```
public Task < UpdateRowResponse > UpdateRowAsync (
    UpdateRowRequest request );
```

Example

Update the data of the specified row.

```
// Define the primary keys of the row, which
// must be consistent with the primary keys defined in
// TableMeta during table creation
PrimaryKey primaryKey = new PrimaryKey ();
primaryKey.Add ("pk0", new ColumnValue ( 0 ));
primaryKey.Add ("pk1", new ColumnValue (" abc "));

// Define the attribute columns of the 100
// rows
UpdateOfAttribute attribute = new UpdateOfAttribute ();
attribute.AddAttributeColumnToPut (" col0 ", new
ColumnValue ( 0 ));
attribute.AddAttributeColumnToPut (" col1 ", new
ColumnValue (" b ")); // Change the original value ' a '
// to ' b '
attribute.AddAttributeColumnToPut (" col2 ", new
ColumnValue ( true ));

try
{
    // Construct the request object for row
    // updating, with RowExistenceExpectation.IGNORE indicating
    // that data is still updated even when the specified
    // row does not exist
    var request = new UpdateRowRequest ( TableName,
    new Condition ( RowExistenceExpectation.IGNORE ),
    primaryKey, attribute );
    // Call the UpdateRow interface
    otsClient.UpdateRow ( request );

    // Execution is successful if no exception is
    // thrown.
    Console.WriteLine (" Update row succeeded .");
}
catch ( Exception ex )
{
    // Execution fails if an exception is thrown
    // , and an error message is printed .
    Console.WriteLine (" Update row failed , exception
    :{ 0 }", ex . Message );
}
```



Note:

Row updating supports the use of conditional statements.

Code details can be obtained at [UpdateRow@GitHub](#).

DeleteRow

API

```

( delete )/// < summary >( delete )
/// Delete a data row based on the specified
table name and primary keys .
/// < / summary >
/// < param name =" request "> Request instance < / param >
/// < returns > Response instance < / returns >
public DeleteRowResponse DeleteRow ( DeleteRowRequest request );

/// < summary >
/// Asynchronous form of DeleteRow .
/// < / summary >
public Task < DeleteRowResponse > DeleteRowAsync (
DeleteRowRequest request );

```

Example

Delete a data row.

```

// The values of the primary key columns of
the row to be deleted are 0 and " abc ".
var primaryKey = new PrimaryKey ();
primaryKey . Add ( " pk0 ", new ColumnValue ( 0 ));
primaryKey . Add ( " pk1 ", new ColumnValue ( " abc "));

try
{
    // Construct a request , with Condition .
    EXPECT_EXISTS indicating that the row is deleted only
    when it exists
    var deleteRowRequest = new DeleteRowRequest (
SampleTable , Condition . EXPECT_EXISTS , primaryKey );

    // Call the DeleteRow interface to delete the
    row
    otsClient . DeleteRow ( deleteRowRequest );

    // The row is deleted successfully if no
    exception is thrown .
    Console . WriteLine ( " Delete table succeeded .");
}
catch ( Exception ex )
{
    // The row fails to be deleted if an
    exception is thrown , and an error message is printed
    .
    Console . WriteLine ( " Delete table failed ,
exception :{ 0 }", ex . Message );
}

```



Note:

Row deletion supports the use of conditional statements.

Code details can be obtained at [DeleteRow@GitHub](#).

6.6 Multiple-row operations

The Table Store SDK provides the following multi-row operation interfaces:

BatchGetRow, BatchWriteRow, GetRange, and GetRangeIterator.

BatchGetRow

Batch reads several data rows from one or more tables. It is essentially a set of multiple GetRow operations. Each operation is executed, returns results, and consumes capacity units independently.

Compared to the execution of a large number of GetRow operations, the BatchGetRow operation reduces the request response time and increases the data read rate.

API

```

        /// < summary >
        /// < para > The BatchGetRow operation is
        essentially a set of multiple GetRow operations . </
        para >
        /// < para > Each operation is executed , returns
        results , and consumes capacity units independen tly . </
        para >
        /// Compared to the execution of a large number
        of GetRow operations , the BatchGetRow operation
        reduces the request response time and increases the
        data read rate .
        /// </ summary >
        /// < param name =" request "> Request instance </ param >
        /// < returns > Response instance </ returns >
        public BatchGetRowResponse BatchGetRow ( BatchGetRow
        wRequest request );

        /// < summary >
        /// Asynchronous form of BatchGetRow .
        /// </ summary >
        public Task < BatchGetRowResponse > BatchGetRowAsync (
        BatchGetRow wRequest request );

```

Example

Batch read 10 data rows.

```

// Construct a batch - read request object ,
including the primary key values of the 10 rows
List < PrimaryKey > primaryKey s = new List <
PrimaryKey >();
for ( int i = 0 ; i < 10 ; i ++ )
{
    PrimaryKey primaryKey = new PrimaryKey ();
    primaryKey . Add ( " pk0 " , new ColumnValu e ( i ));
    primaryKey . Add ( " pk1 " , new ColumnValu e ( " abc
));
    primaryKey s . Add ( primaryKey );
}

```

```

        try
        {
            BatchGetRow wRequest request = new BatchGetRow
wRequest ();
            request . Add ( TableName , primaryKey s );

            // Call BatchGetRow w to read 10 data rows
            var response = otsClient . BatchGetRow w ( request );
            var tableRows = response . RowDataGroupByTable ;
            var rows = tableRows [ TableName ];

            // Input the data of the rows . The input
data is omitted here . For more information , see the
GitHub link .

            // BatchGetRow w may partially fail . Therefore ,
the status of each row must be checked . For more
information , see the GitHub link .
        }
        catch ( Exception ex )
        {
            // Execution fails if an exception is thrown
, and an error message is printed .
            Console . WriteLine ( " Batch get row failed ,
exception :{ 0 }" , ex . Message );
        }
    }

```

**Note:**

BatchGetRow supports filter using conditional statements.

Code details can be found at [BatchGetRow@GitHub](#).

BatchWriteRow

BatchWriteRow inserts, modifies, or deletes several data rows in one or more tables. It is a set of multiple **PutRow**, **UpdateRow**, and **DeleteRow** operations. Each operation is executed, returns results independently, and consumes capacity units independently.

API

```

    /// < summary >
    /// < para > Batch inserts , modifies , or deletes
several data rows in one or more tables . < / para >
    /// < para > The BatchWrite Row operation is
essentially a set of multiple PutRow , UpdateRow ,
and DeleteRow operations . Each operation is executed ,
returns results , and consumes capacity units independen
tly . < / para >
    /// < para > Compared to the execution of a
large number of write operations , the BatchWrite Row
operation reduces the request response time and
increases the data write rate . < / para >
    /// < / summary >
    /// < param name =" request "> Request instance < / param >
    /// < returns > Response instance < / returns >

```

```

    public BatchWriteRowResponse BatchWriteRow (
        BatchWriteRowRequest request );

    /// < summary >
    /// Asynchronous row of BatchWriteRow
    /// < / summary >
    /// < param name =" request ">< / param >
    /// < returns >< / returns >
    public Task < BatchWriteRowResponse > BatchWriteRowAsync ( BatchWriteRowRequest request );

```

Example

Batch import 100 data rows.

```

// Construct a batch - import request object ,
// including the primary key values of the 100 rows
var request = new BatchWriteRowRequest ();
var rowChanges = new RowChanges ();
for ( int i = 0 ; i < 100 ; i ++ )
{
    PrimaryKey primaryKey = new PrimaryKey ();
    primaryKey . Add ( " pk0 " , new ColumnValue ( i ));
    primaryKey . Add ( " pk1 " , new ColumnValue ( " abc
"));

    // Define the attribute columns of the 100
rows
    UpdateOfAttribute attribute = new UpdateOfAttribute ();
    attribute . AddAttributeColumnToPut ( " col0 " , new
ColumnValue ( 0 ));
    attribute . AddAttributeColumnToPut ( " col1 " , new
ColumnValue ( " a " ));
    attribute . AddAttributeColumnToPut ( " col2 " , new
ColumnValue ( true ));

    rowChanges . AddUpdate ( new Condition ( RowExistenceExpectation . IGNORE ) , primaryKey , attribute );
}

request . Add ( TableName , rowChanges );

try
{
    // Call BatchWriteRow
    var response = otsClient . BatchWriteRow ( request );

    var tableRows = response . TableResponses ;
    var rows = tableRows [ TableName ];

    // Partial operations of BatchGetRow may fail
; therefore , the status of each row must be checked
. For more information , see the GitHub link .
}
catch ( Exception ex )
{
    // Execution fails if an exception is thrown
, and an error message is printed .
    Console . WriteLine ( " Batch put row failed ,
exception : { 0 } " , ex . Message );
}

```

```
}
```

**Note:**

BatchWriteRow supports filter using conditional statements.

Code details can be found at [BatchWriteRow@GitHub](#).

GetRange

Reads data in the specified primary key range.

API

```

    /// < summary >
    /// Get data from multiple rows in the
    specified range .
    /// < / summary >
    /// < param name = " request "> Request instance < / param >
    /// < returns > Response instance < / returns >
    public GetRangeResponse GetRange ( GetRangeRequest request );

    /// < summary >
    /// Asynchronous form of GetRange .
    /// < / summary >
    /// < param name = " request ">< / param >
    /// < returns >< / returns >
    public Task < GetRangeResponse > GetRangeAsync (
        GetRangeRequest request );

```

Example

Read data in the specified range.

```

    // Read all rows in the range from ( 0 ,
    INF_MIN ) to ( 100 , INF_MAX )
    var inclusiveStartPrimaryKey = new PrimaryKey ();
    inclusiveStartPrimaryKey.Add ( " pk0 ", new
    ColumnValue ( 0 ));
    inclusiveStartPrimaryKey.Add ( " pk1 ", ColumnValue .
    INF_MIN );

    var exclusiveEndPrimaryKey = new PrimaryKey ();
    exclusiveEndPrimaryKey.Add ( " pk0 ", new ColumnValue
    ( 100 ));
    exclusiveEndPrimaryKey.Add ( " pk1 ", ColumnValue .
    INF_MAX );

    try
    {
        // Constructs a read request object in the
        specified range
        var request = new GetRangeRequest ( TableName ,
        GetRangeDirection . Forward ,
        inclusiveStartPrimaryKey ,
        exclusiveEndPrimaryKey );

        var response = otsClient . GetRange ( request );
    }

```

```

        // Continue the read operation if only
        partial data is returned
        var rows = response . RowDataList ;
        var nextStartPrimaryKey = response . NextPrimary
        yKey ;
        while ( nextStartPrimaryKey != null )
        {
            request = new GetRangeRequest ( TableName ,
            GetRangeDirection . Forward ,
            nextStartPrimaryKey , exclusiveEndPrimary
            Key );
            response = otsClient . GetRange ( request );
            nextStartPrimaryKey = response . NextPrimary
            yKey ;
            foreach ( RowDataFromGetRange row in
            response . RowDataList )
            {
                rows . Add ( row );
            }
        }

        // Output the data of the rows . The output
        data is omitted here . For more information , see
        the GitHub link .

        // Execution is successful if no exception is
        thrown .
        Console . WriteLine ( " Get range succeeded " );
    }
    catch ( Exception ex )
    {
        // Execution fails if an exception is thrown
        , and an error message is printed .
        Console . WriteLine ( " Get range failed , exception
        :{ 0 }" , ex . Message );
    }

```

**Note:**

GetRange supports filter using conditional statements.

Code details can be found at [GetRange@GitHub](#).

GetRangeIterator

Gets the iterator in the specified range.

API

```

    /// < summary >
    /// Get data from multiple rows in the
    specified range . The system returns the iterator used
    to iterate through each data row .
    /// < / summary >
    /// < param name = " request "> see cref = " GetIterato
    rRequest ">< / param >
    /// < returns > Return the < see cref = " RowDataFro
    mGetRange "> iterator . < / returns >

```

```
public IEnumerable< RowDataFromGetRange > GetRangeIterator ( GetIteratorRequest request );
```

Example

Get an iterator.

```
// Read all rows in the range from ( 0 , " a " )
to ( 1000 , " xyz " )
PrimaryKey inclusiveStartPrimaryKey = new
PrimaryKey ();
    inclusiveStartPrimaryKey . Add ( " pk0 " , new
ColumnValue ( 0 ));
    inclusiveStartPrimaryKey . Add ( " pk1 " , new
ColumnValue ( " a " ));

PrimaryKey exclusiveEndPrimaryKey = new PrimaryKey
();
    exclusiveEndPrimaryKey . Add ( " pk0 " , new ColumnValue
( 1000 ));
    exclusiveEndPrimaryKey . Add ( " pk1 " , new ColumnValue
( " xyz " ));

// Construct a CapacityUnit to record the CUs
consumed by iteration
var cu = new CapacityUnit ( 0 , 0 );

try
{
    // Construct a GetIteratorRequest . Filter
criteria are supported .
    var request = new GetIteratorRequest ( TableName
, GetRangeDirection . Forward , inclusiveStartPrimaryKey ,
exclusiveEndPrimaryKey , cu );

    var iterator = otsClient . GetRangeIterator (
request );
    // Iterator that reads data in a traversal
method
    foreach ( var row in iterator )
    {
        // Processing logic
    }

    Console . WriteLine ( " Iterate row succeeded " );
}
catch ( Exception ex )
{
    Console . WriteLine ( " Iterate row failed ,
exception :{ 0 }", ex . Message );
}
```



Note:

GetRangeIterator supports filter using conditional statements.

Code details can be found at [GetRangeIterator@GitHub](#).

6.7 Error handling

Method

Currently, the Table Store C# SDK adopts the `Exception` method to handle errors. Operation is successful if the called interface does not throw an exception. If an exception is thrown, operation fails.



Note:

Batch operation interfaces such as `BatchGetRow` and `BatchWriteRow` are successfully called only when the system checks that the status of each row is successful.

Exception

The Table Store C# SDK has two types of exceptions: `OTSClientException` and `OTSServerException` inherited from `Exception`.

- `OTSClientException`: indicates an internal SDK exception, for example, incorrect parameter values or failure to return parsed results.
- `OTSServerException`: indicates a server error generated by parsing a server error message. `OTSServerException` has the following components:
 - `HttpStatusCode`: a returned HTTP code, for example, 200 or 404.
 - `ErrorCode`: an error type string returned by Table Store.
 - `ErrorMessage`: an error message string returned by Table Store.
 - `RequestId`: the UUID that uniquely identifies a request. If a problem persists, save the `RequestId` and [open a ticket](#).

Retry

The system retries the operation when an error occurs in the SDK. By default, the operation is retried three times at a maximum interval of two seconds. For more information, see the `Aliyun.OTS.Retry.DefaultRetryPolicy` class.

You can use `RetryPolicy` in `OTSClientConfig` to customize a retry policy.

7 C++ SDK

7.1 Preface

This document describes how to install and use Table Store C++ SDK. It is applicable to version 4.0.0 and later. It is assumed that you have signed up for the Table Store service and created an AccessKeyId and AccessKeySecret.

- If you have not yet signed up or want to learn more about the Alibaba Cloud Table Store service, visit the [Table Store product homepage](#) for more information.
- If you have not yet created the AccessKeyId and AccessKeySecret, go to the [AccessKey console](#) and create an AccessKey.

Download address: [GitHub](#)

7.2 Installation

C++ is special, so we recommend that you use the methods described in this topic to compile and copy `build / release / src / tablestore / core / impl / buildinfo . cpp` for backup. Then copy the C++ SDK source code and `buildinfo . cpp` to your own code library and compile it using your compilation system.

Compilation parameters

When compiling the client-side code, some compiler behaviors must be ensured. This means that some compiler parameters are required.

The GCC compiler parameters are described as follows.

Parameter	Required	Description
<code>--std=gnu++03</code>	Yes	Supports C++98 TR1 language with GCC extension (that is, typeid).
<code>-pthread</code>	Yes	Required parameter for multithreaded programming. This parameter is required for both compiling and linking.

Parameter	Required	Description
-fwrapv	Yes	Rotate upon integer overflow. Specifically, this yields 0 in the case of an unsigned integer overflow and the smallest negative number in the case of a signed integer overflow. The client checks overflow based on this behavior.
-O2	Recommended	Optimization grade. In general, a higher optimization grade is not recommended.
-fsanitize=address and-fvar-tracking-assignments	Recommended	Later versions than gcc-4.9 support libasan, a lightweight detector that can quickly detect errors in memory usage. If you need Table Store developers to locate errors, please take these two compilation errors so that the error can be reproduced. You must also take the previous parameter when linking.  Notice: libasan and valgrind are incompatible.

Environment dependency and pre-compiled packages

The Debian 8 system is used here as an example to demonstrate how to generate pre-compiled packages.

1. Open the `docker / debian8 / Dockerfile` file and select `dockerfile` to export the environment dependency of the client on the system. This method automatically guarantees the consistency between the code and the environment.

```
RUN apt - get install - y scons g ++ libboost - all - dev
    protobuf - compiler libprotobu f - dev uuid - dev libssl -
dev
RUN apt - get install - y ca - certificat es # for
HTTPS support
```

The SDK depends on the following:

- `scons` & `gcc`: Compilation system
 - `boost`
 - `uuid`
 - `protobuf`: serialization library
 - `openssl`: signature, used to support HTTPS
 - `ca-certificates`: used to support HTTPS only. If you only use the Table Store HTTP address, you can choose not to install this library. We recommend that you use the more secure HTTPS.
2. You can compile the client after installing these packages. Download the source code of the client and run `scons` in the source code directory.

```
$ git clone https :// github . com / aliyun / aliyun -
tablestore - cpp - sdk . git
$ cd aliyun - tablestore - cpp - sdk
$ scons - j4
```

Once these steps have been completed, a tar package is compiled. Typically, you can find the package name in the final output of `scons`. For example, the package name and path of the Debian 8 system are: `build / release / pkg / aliyun - tablestore - cpp98 - sdk - 4 . 4 . 1 - debian8 . 9 - x86_64 . tar . gz`.

- The package name includes the following elements:
 - C++ version (C++98)
 - SDK version (4.4.1)
 - OS version (Debian 8.9)
 - OS architecture (x86_64)

- The contents of the package include:

```
$ tar -tf build / release / pkg / aliyun - tablestore - cpp98 -
sdk - 4 . 4 . 1 - debian8 . 9 - x86_64 . tar . gz
version . ini
lib / libtablest ore_core . so
lib / libtablest ore_core_s tatic . a
lib / libtablest ore_util . so
lib / libtablest ore_util_s tatic . a
include / tablestore / util / arithmetic . hpp
include / tablestore / util / assert . hpp
include / tablestore / util / foreach . hpp
include / tablestore / util / iterator . hpp
include / tablestore / util / logger . hpp
include / tablestore / util / logging . hpp
include / tablestore / util / mempiece . hpp
include / tablestore / util / mempool . hpp
include / tablestore / util / metaprogra mming . hpp
include / tablestore / util / move . hpp
include / tablestore / util / optional . hpp
include / tablestore / util / prettyprin t . hpp
include / tablestore / util / random . hpp
include / tablestore / util / result . hpp
include / tablestore / util / security . hpp
include / tablestore / util / seq_gen . hpp
include / tablestore / util / threading . hpp
include / tablestore / util / timestamp . hpp
include / tablestore / util / try . hpp
include / tablestore / util / assert . ipp
include / tablestore / util / iterator . ipp
include / tablestore / util / logging . ipp
include / tablestore / util / mempiece . ipp
include / tablestore / util / move . ipp
include / tablestore / util / prettyprin t . ipp
include / tablestore / core / client . hpp
include / tablestore / core / error . hpp
include / tablestore / core / range_iter ator . hpp
include / tablestore / core / retry . hpp
include / tablestore / core / types . hpp
```

The package currently includes the following elements:

- Version file: `version . ini`
- Library files: all files under `lib /`. Where `libtablest ore_core_s tatic . a` depends on `libtablest ore_util_s tatic . a`. The Moments library is similar.
- Header files: all files under `include /`.

7.3 Initialization

The Table Store client provides callers with a series of methods for operating tables, single rows of data, multiple rows of data, and so on.

Determine an endpoint

Endpoint is the domain name address for the Alibaba Cloud Table Store service in each region. Current supported formats include:

Example	Description
<code>http://sun.cn-hangzhou-ots.aliyuncs.com</code>	Public network access to the sun instance in Hangzhou using HTTP.
<code>https://sun.cn-hangzhou-ots.aliyuncs.com</code>	Public network access to the sun instance in Hangzhou using HTTPS.



Note:

- Table Store supports public and private network access.
- You can log on to the [Table Store console](#) and go to the instance details page. The Instance Access URL is the endpoint for that instance.

Configure an AccessKey

To access the Alibaba Cloud Table Store service, you must have a valid AccessKey for signature authentication. These three ways of configuring an AccessKey are currently supported:

- Use the AccessKeyId and AccessKeySecret of your primary account. The procedure is as follows:
 1. Register an [Alibaba Cloud account](#).
 2. Log on to the [AccessKey console](#) to create an AccessKeyId and AccessKeySecret.
- Use the AccessKeyId and AccessKeySecret of the RAM user authorized to access Table Store. The procedure is as follows:
 1. Use your primary account to access [Resource Access Management \(RAM\)](#) and create a new RAM user or use an existing one.
 2. Use your primary account to authorize the RAM user to access Table Store.
 3. You can then use the AccessKeyId and AccessKeySecret of the authorized RAM user to access Table Store.

- Use an STS token for temporary access. The procedure is as follows:
 1. The application server accesses the RAM/STS service, obtains a temporary AccessKeyId, AccessKeySecret, and token, and sends them to you.
 2. You can use the temporary key to access the Table Store service.

Create a client

When you use a Table Store SDK, you first have to construct a client and then call its interface to access the Table Store service. The client interface is consistent with the RESTful API provided by Table Store.

Client type

Table Store C++ SDK offers two types of client: SyncClient and AsyncClient. These correspond to the synchronous interface and asynchronous interface respectively.

- Synchronous interface: As soon as the call is completed, the request is executed, making it quite a convenient method. You can use the synchronous interface to learn more about the various Table Store functions.
- Asynchronous interface: These provide more flexibility than synchronous interfaces. If you have high performance requirements, you can make a trade-off between using asynchronous interfaces and multithreading.



Note:

Both SyncClient and AsyncClient are thread-safe and can automatically and internally manage threads and connection resources. You do not have to create a client for every thread or request. Instead, create a global client.

Synchronous interface

- Create directly (use a Table Store endpoint to create a client)

```
Endpoint ep ("YourEndpoint", "YourInstance");
Credential cr ("AccessKeyId", "AccessKeySecret");
ClientOptions opts;
SyncClient * client = NULL;
Optional<OTSError> res = SyncClient::create(client, ep, cr, opts);
```



Note:

Avoid using the AccessKey of your primary account to access Table Store. We recommend using a temporary token or the AccessKey of a sub-account. If you use a temporary STS token, the credential object in the aforementioned code must be

```
changed to: Credential cr (" AccessKeyId ", " AccessKeySecret ",
" SecurityToken ");.
```

The configuration items included in ClientOptions are described as follows. You can use the default values or define them yourself.

mMaxConnections	The maximum number of total connections and also the maximum number of concurrent requests. A persistent connection is maintained between the SDK and the Table Store service. Each new request is sent by a randomly selected idle connection.
mConnectTimeout	Connection timeout. Considering the DNS resolution time, the recommended connection timeout is 10 seconds at least.
mRequestTimeout	Request timeout time.
mRetryStrategy	Retry strategy. By default, the retry strategy retries a failed idempotence request within 10 seconds. You can define your own retry strategy.
mLogger	Logger. By default, the logger exports logs to standard errors. We recommend you define your own logger.
mActors	Thread pool. The thread pool used to run your callback. The default is 10 threads.

- Create from AsyncClient

```
AsyncClient & async = ... ;
SyncClient * sync = SyncClient :: create ( async );
```

Asynchronous interface

See [Asynchronous interfaces](#) for more information on how to create and use an asynchronous interface client.

Multithreading

Multithreading is supported. When multithreading, we recommend sharing one client object.

7.4 Table-level operations

The SDK offers table operation interfaces, such as CreateTable, ListTable, UpdateTable, DescribeTable, and DeleteTable.



Note:

The following operation examples are based on synchronous interfaces. See [Asynchronous interfaces](#) for more information on how to operate asynchronous interfaces.

Create a table (CreateTable)

You must always specify the name and primary key of the table you want to create. The primary key contains between 1 and 4 primary key columns, each of which has a name and type.

In Table Store tables, you can set auto increments in primary key columns. See [Primary key column auto-increment](#) for more information.

Example

In this example, the table name is `simple_create_delete_table`. The primary key only has one primary key column, named `pkey`, and is of an integer type (`kPKT_Integer`).

```
CreateTable eRequest req ;
{
    // immutable configurations of the table
    TableMeta & meta = req.mutableMeta();
    meta.mutableTableName() = "simple_create_delete_table";
    {
        // with exactly one integer primary key column
        Schema & schema = meta.mutableSchema();
        PrimaryKeyColumnSchema & pkColSchema = schema.append();
        pkColSchema.mutableName() = "pkey";
        pkColSchema.mutableType() = kPKT_Integer;
    }
}
CreateTable eResponse resp ;
Optional<OTSError> res = client.createTable(resp, req);
```



Note:

Detailed code is available at [createTable@GitHub](#).

Variable parameters

You can set several variable parameters for the data table. Variable parameters can be set when creating tables, or changed using [Update a table \(UpdateTable\)](#).

Variable parameters include the following:

Variable parameter	Name	Default value
<code>mutableTimeToLive()</code>	Data versions and time to live	-1 (meaning data never expires)
<code>mutableMaxVersions()</code>	Data versions and time to live	1
<code>mutableMaxTimeDeviation()</code>	Data versions and time to live	86,400s (or one day)
<code>mutableReservedThroughput()</code>	Read/write throughput	0 (all read/write charged on a Pay-As-You-Go basis)

The following is an example of setting the reserved read/write throughput when creating a table:

```

CreateTableRequest req ;
{
    // immutable configurations of the table
    TableMeta & meta = req.mutableMeta();
    meta.mutableTableName() = "create_table_with_reserved_throughput";
    {
        // with exactly one integer primary key column
        Schema & schema = meta.mutableSchema();
        PrimaryKeyColumnSchema & pkColSchema = schema.append();
        pkColSchema.mutableName() = "pkey";
        pkColSchema.mutableType() = kPKT_Integer;
    }
}
{
    TableOptions & opts = req.mutableOptions();
    {
        // 0 reserved read capacity-unit, 1 reserved write capacity-unit
        CapacityUnit cu(0, 1);
        opts.mutableReservedThroughput().reset(util::move(cu));
    }
}
CreateTableResponse resp ;
Optional<OTSError> res = client.createTable(resp, req);

```

List a table name (ListTable)

Used to get the names of all tables created in the current instance.

API

`SyncClient::listTable()` is used to list all tables under the instance.

```
SyncClient * client = ... ;
ListTableRequest req ;
ListTableResponse resp ;
Optional < OTSError > res = client -> listTable ( resp , req );
```

Example

Used to get the names of all tables under the instance.

```
const IVector < string >& xs = resp . tables ();
for ( int64_t i = 0 ; i < xs . size (); ++ i ) {
    cout << xs [ i ] << endl ;
}
```



Note:

Detailed code is available at [listTable@GitHub](#).

Update a table (UpdateTable)

Updates the variable parameters of a specified table.

Example

Update the reserved throughput.

```
UpdateTableRequest req ;
req . mutableTable () = " YourTable ";
UpdateTableResponse resp ;
{
    TableOptions & opts = req . mutableOptions ();
    {
        // 0 reserved read capacity - unit , 1 reserved
        write capacity - unit
        CapacityUnit cu ( 0 , 1 );
        opts . mutableReservedThroughput (). reset ( util :: move
        ( cu ));
    }
}
Optional < OTSError > res = client . updateTable ( resp , req );
```



Note:

Detailed code is available at [updateTable@GitHub](#).

Query the table information (DescribeTable)

You can query the following table information through the `describeTable()` interface:

Information item	Description
Table status	Includes: • <code>kTS_Active</code>: The table can offer normal read and write services. • <code>kTS_Inactive</code>: The table cannot be read or written, but table data is reserved. This status usually occurs during a primary-backup table switch. • <code>kTS_Loading</code>: The table is being created. The table cannot be read or written. • <code>kTS_Unloading</code>: The table is being deleted. The table cannot be read or written. • <code>kTS_Updating</code>: The variable table parameters are being updated. The table cannot be read or written.
Table meta	See Create a table (CreateTable) .
Variable table parameters	See variable parameters.
Split points between shards	A Table Store table is split horizontally into several shards. Split points can be obtained through this interface.  Note: Table Store can be automatically split and merged in the background according to the load. As such, the split points received by this interface are guaranteed to reflect past shards, but not necessarily match what is currently going on.

Example

```
DescribeTableRequest req;  
req.mutableTable() = "YourTable";  
DescribeTableResponse resp;
```

```
Optional < OTSError > res = client . describeTa ble ( resp , req
);
```



Note:

Detailed code is available at [describeTable@GitHub](#).

Delete a table (DeleteTable)

Deletes specified tables under this instance. The only requirement is to specify the table name.

Example

```
DeleteTabl eRequest req ;
req . mutableTab le () = " YourTable ";
DeleteTabl eResponse resp ;
Optional < OTSError > res = client . deleteTabl e ( resp , req
);
```



Note:

Detailed code is available at [deleteTable@GitHub](#).

7.5 Single-row operations

Table Store offers interfaces for single-row operations, such as PutRow, GetRow, UpdateRow, and DeleteRow.



Note:

The following operation examples are based on synchronous interfaces. See [Asynchronous interfaces](#) for more information on how to operate asynchronous interfaces.

Put in a row of data (PutRow)

The PutRow interface is used to insert a row of data. The PutRow interface is used to insert a row of data. If the row does not exist, this operation inserts a row. If the row exists, this operation overwrites it. (This means that all columns and column values of all versions of the original row are deleted).

Example

```
PutRowRequ est req ;
{
    RowPutChan ge & chg = req . mutableRow Change ();
```

```

    chg.mutableTable() = "YourTable";
    {
        // set primary key of the row to put
        PrimaryKey & pkey = chg.mutablePrimaryKey();
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("pkey - value"));
    }
    {
        // set attributes of the row to put
        IVector<Attribute>& attrs = chg.mutableAttributes();
        attrs.append() = Attribute(
            "attr",
            AttributeValue::toInteger(123));
    }
}
PutRowResponse resp;
Optional<OTSError> res = client.putRow(resp, req);

```

Get the data of a row (GetRow)

The GetRow interface is used to read the data of a single row.

Specify the table name and the primary key of a row. The two possible reading results are as follows:

- If this row exists, the GetRowResponse object returns each primary key column and attribute column for the row.
- If this row does not exist, the GetRowResponse object does not include rows and it does not report errors.

Example

```

GetRowRequest req;
{
    PointQueryCriterion & query = req.mutableQueryCriterion();
    query.mutableTable() = "YourTable";
    {
        PrimaryKey & pkey = query.mutablePrimaryKey();
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("some_key")); // Assume
the table has only one primary key column that is
of the string type.
    }
    query.mutableMaxVersions().reset(1);
}
GetRowResponse resp;
Optional<OTSError> res = client.getRow(resp, req);

```

Update the data of a row (UpdateRow)

The UpdateRow interface is used to update a row of data. If the row does not exist, a new row is written.

The update operation has the following four possible scenarios:

- Writing a column value without specifying the version number. The Table Store service automatically offers a version number to make sure that the version number is incremented.
- Writing a column value with a specified version number. If the column does not have the column value of this version, the data is inserted. If the column does have it, the original value is overwritten.
- Delete the column values of the specified version.
- Delete the column values of all versions for the entire column.

Example

```
UpdateRowRequest req ;
{
    RowUpdateChange & chg = req.mutableRowChange();
    chg.mutableTable() = "YourTable";
    {
        // set primary key of the row to put
        PrimaryKey & pkey = chg.mutablePrimaryKey();
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("pkey"));
    }
    {
        // insert a value without specifying version
        RowUpdateChange::Update & up = chg.mutableUpdates()
        .append();
        up.mutableType() = RowUpdateChange::Update::kPut;
        up.mutableAttributeName() = "attr0";
        up.mutableAttributeValue().reset(AttributeValue::toStr(
            "new value without specifying version"));
    }
    {
        // insert a value with version
        RowUpdateChange::Update & up = chg.mutableUpdates()
        .append();
        up.mutableType() = RowUpdateChange::Update::kPut;
        up.mutableAttributeName() = "attr1";
        up.mutableAttributeValue().reset(AttributeValue::toStr(
            "new value with version"));
        up.mutableTimestamp().reset(UtcTime::now());
    }
    {
        // delete a value with specific version
        RowUpdateChange::Update & up = chg.mutableUpdates()
        .append();
        up.mutableType() = RowUpdateChange::Update::kDelete;
        up.mutableAttributeName() = "attr2";
        up.mutableTimestamp().reset(UtcTime::now());
    }
    {
        // delete all values of a attribute column
```

```

        RowUpdateC hange :: Update & up = chg.mutableUpdates
().append();
        up.mutableType() = RowUpdateC hange :: Update ::
kDeleteAll;
        up.mutableAttributeName() = "attr3";
    }
}
UpdateRowResponse resp;
Optional<OTSError> res = client.updateRow(resp, req);

```

Delete the data of a row (DeleteRow)

The DeleteRow interface is used to delete a row. No error is reported, regardless of whether this row exists or not.

Example

```

DeleteRowRequest req;
{
    RowDeleteC hange & chg = req.mutableRowChange();
    chg.mutableTable() = "YourTable";
    {
        // set primary key of the row to delete
        PrimaryKey & pkey = chg.mutablePrimaryKey();
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toInteger(1));
    }
}
DeleteRowResponse resp;
Optional<OTSError> res = client.deleteRow(resp, req);

```

7.6 Multiple-row operations

Table Store offers interfaces for multi-row operations such as BatchGetRow, BatchWriteRow, and GetRange.



Note:

The following operation examples are based on synchronous interfaces. See [Asynchronous interfaces](#) for more information on how to operate asynchronous interfaces.

BatchGetRow

Batch reads several rows of data from one or more tables.

BatchGetRow can be seen as a set of multiple GetRow operations. Performing each operation, returning the results, and calculating the service capacity unit are all done independently. In comparison with performing a large number of GetRow operations

, using `BatchGetRow` can effectively shorten the request response time and increase the data read rate.

Two possible errors may occur when using `BatchGetRow`:

- Overall request error, such as a network error. These errors are stored in the return values of `batchGetRow()`.
- No overall request error, but some individual rows contain errors. These errors are stored in the results of the corresponding rows.

Example

```
BatchGetRow wRequest req ;
{
    MultiPoint QueryCriterion & criterion = req.mutableCriteria().append();
    IVector< MultiPoint QueryCriterion::RowKey> & rowkeys = criterion.mutableRowKeys();
    {
        MultiPoint QueryCriterion::RowKey & exist = rowkeys.append();
        exist.mutableGet().append() = PrimaryKeyColumn("pkey",
            PrimaryKeyValue::toStr("some key"));
        exist.mutableUserData() = &userDataForSomeKey;
    }
    {
        MultiPoint QueryCriterion::RowKey & exist = rowkeys.append();
        exist.mutableGet().append() = PrimaryKeyColumn("pkey",
            PrimaryKeyValue::toStr("another key"));
        exist.mutableUserData() = &userDataForAnotherKey;
    }
    criterion.mutableTable() = "YourTable";
    criterion.mutableMaxVersions().reset(1);
}
BatchGetRow wResponse resp ;
Optional< OTSError> res = client.batchGetRow(resp, req);
```



Note:

Detailed code is available at [batchGetRow@GitHub](#).

BatchWriteRow

Batch insert, modify, or delete several rows of data from one or more tables.

`BatchWriteRow` can be seen as a set of multiple `PutRow`, `UpdateRow`, and `DeleteRow` operations. Performing each operation, returning the results, and calculating the service capacity unit are all done independently.

Two possible errors may occur when using `BatchWriteRow`:

- Overall request error, such as a network timeout. These errors are stored in the return values of `batchWriteRow()`.
- Errors in individual rows, such as an invalid primary key value. These errors are stored in every row of `BatchWriteRowResponse`.

Example

```
static const char kPutRow [] = " PutRow ";
static const char kUpdateRow [] = " UpdateRow ";
static const char kDeleteRow [] = " DeleteRow ";

BatchWrite RowRequest req ;
{
    // put row
    BatchWrite RowRequest :: Put & put = req . mutablePut s ().
append ();
    put . mutableUse rData () = kPutRow ;
    put . mutableGet (). mutableTab le () = kTableName ;
    PrimaryKey & pkey = put . mutableGet (). mutablePri maryKey
();
    pkey . append () = PrimaryKey Column (
        " pkey ",
        PrimaryKey Value :: toStr (" row to put "));
}
{
    // update row
    BatchWrite RowRequest :: Update & update = req . mutableUpd
ates (). append ();
    update . mutableUse rData () = kUpdateRow ;
    update . mutableGet (). mutableTab le () = kTableName ;
    PrimaryKey & pkey = update . mutableGet (). mutablePri
maryKey ();
    pkey . append () = PrimaryKey Column (
        " pkey ",
        PrimaryKey Value :: toStr (" row to update "));
    RowUpdateC hange :: Update & attr = update . mutableGet ().
mutableUpd ates (). append ();
    attr . mutableTyp e () = RowUpdateC hange :: Update :: kPut ;
    attr . mutableAtt rName () = " attr0 ";
    attr . mutableAtt rValue (). reset ( AttributeV alue :: toStr
(" some value "));
}
{
    // delete row
    BatchWrite RowRequest :: Delete & del = req . mutableDel
etes (). append ();
    del . mutableUse rData () = kDeleteRow ;
    del . mutableGet (). mutableTab le () = kTableName ;
    PrimaryKey & pkey = del . mutableGet (). mutablePri maryKey
();
    pkey . append () = PrimaryKey Column (
        " pkey ",
        PrimaryKey Value :: toStr (" row to delete "));
}
BatchWrite RowRespons e resp ;
```

```
Optional < OTSError > res = client . batchWrite Row ( resp , req );
```

**Note:**

Detailed code is available at [batchWriteRow@GitHub](#).

GetRange

Read data within the specified primary key range.

`RangeIterator` is recommended. To construct `RangeIterator`, you must provide:

- [AsyncClient](#).
- `RangeQueryCriterion` is similar to `PointQueryCriterion`, but `RangeQueryCriterion` also requires:
 - Set the starting point (inclusive) and ending point (exclusive) of the range. In addition to the normal primary key value, you can also use two special values: “negative infinity” (strictly less than all normal primary key column values) and “positive infinity” (strictly greater than all primary key column values).
 - Set the reading to positive sequence (values going from small to large) or reverse sequence (values going from large to small). The default setting is positive sequence. When reading in positive sequence, the starting point must be smaller than the ending point. In the reverse sequence, the starting point must be greater than the ending point.

The `RangeIterator` object is an iterator that provides three interfaces:

- `moveNext ()` moves the `RangeIterator` object to the next row. A newly constructed `RangeIterator` object must call `moveNext ()` to get the value. If reading data fails, `moveNext ()` displays the error in the return value.
- `valid ()` shows whether the `RangeIterator` object has reached the ending point of the range.
- If `valid ()` is true, row objects can be read using `get ()`. You can avoid memory replication by removing the contents of the row object returned by `get ()`. After removing the content, the directly following `get ()` returns the removed content.

Example

```
RangeQuery Criterion query ;
query . mutableTable () = " YourTable ";
query . mutableMax Versions (). reset ( 1 );
{
```

```

    PrimaryKey & start = query.mutableInclusiveStart();
    start.append() = PrimaryKeyColumn(
        "pkey",
        PrimaryKeyValue::toInfMin());
}
{
    PrimaryKey & end = query.mutableExclusiveEnd();
    end.append() = PrimaryKeyColumn(
        "pkey",
        PrimaryKeyValue::toInfMax());
}
auto_ptr<AsyncClient> aclient(AsyncClient::create(
    client));
RangeIterator iter(*aclient, query);
for(;;) {
    Optional<OTSError> err = iter.moveToNext();
    if (err.present()) {
        // do something with err
        abort();
    }
    if (!iter.valid()) {
        break;
    }
    Row & row = iter.get();
    // do something with row
}

```

**Note:**

Detailed code is available at [GetRange@GitHub](#).

Read specified columns

Table Store supports infinite row width. Typically, reading the entire row is not required, as reading specified columns is enough. `QueryCriterion` (`PointQueryCriterion`, `MultiPointQueryCriterion`, and `RangeQueryCriterion`) provides `mutableColumnsToGet()` to specify columns that must be read. They can be attribute columns or primary key columns. If the value is empty, the entire row is read.

If a specified column does not exist on the read row, the column is missing from the returned result. Table Store does not provide placeholders.

In `GetRange`, if all specified columns are attribute columns, and a row in the range misses all specified columns, this row does not appear in the result. If this row must be noticed, add the primary key column to the specified columns.

Read specified versions

Each attribute column can contain multiple versions, with each version number (timestamp) corresponding to a column value. When reading, you can define the

number of versions (`mutableMaxVersions()`) and the range of versions (`mutableTimeRange()`) to be read.

At least one of the maxversions and the range of versions must be defined.

- If you only define the number of versions, the defined amount of data in all versions is returned, from the newest to the oldest.
- If you only define the range of versions, all data within this range is returned.
- If you define both the number of versions and the range of versions, the defined amount of data within the version range is returned, from the newest to the oldest.

Conditional write

In conditional writing, certain conditions have to be checked and met before rows can be written. Table Store guarantees the atomicity of the condition checks and writing.

Table Store supports row existence conditions and column value conditions:

- Row existence conditions can be divided into the following types:
 - Ignore: This is the default setting whether or not a row exists.
 - ExpectExist: expected to exist. Write if a row exists.
 - ExpectNotExist: expected not to exist. Write if a row does not exist.
- The column value condition is equivalent to the [Filter](#).

7.7 Filter

Table Store can re-filter the read results on the service side to limit the amount of data transmitted on the network.

The internal node of the tree-structured filter is a logical operator (`CompositeColumnCondition`) and its leaf nodes are comparison operators (`SingleColumnCondition`).

- `CompositeColumnCondition` supports `AND`, `OR`, and `NOT`. Two or more sub-nodes can be mounted on `AND` and `OR`. Only one node can be mounted on `NOT`.
- `SingleColumnCondition` supports all 6 comparison conditions (`EQUAL TO`, `NOT EQUAL TO`, `GREATER THAN`, `LESS THAN`, `GREATER THAN OR EQUAL TO`, `LESS THAN OR EQUAL TO`).

- Each `SingleColumnCondition` object supports comparing a column (which can be a primary key column) with a constant. Comparing two columns or two constants is not supported.
- The `latestVersionOnly` parameter of `SingleColumnCondition` controls how multiple versions of the column value are involved in the comparison. The default value is “true” .
 - If true, only the latest version of the column value, falling within the specified range of versions, is involved in the comparison. (These column values are only involved in the comparison. If the filter accepts this row, other versions of the column value are still returned.)
 - If false, when any column value satisfies the conditions, the node considers it true.
- The `passIfMissing` parameter of `SingleColumnCondition` controls what values the node is to consider if the column value is missing. The default value is “false” .

The filter acts on the results of other conditional filters. Therefore, if a column in the filter is not specified in the specified column read, or if no column value within the range of versions exists, the filter considers this column to be missing.

Example

The following example filters out from the table all rows that meet this condition: the primary key column `pkey` is greater than 1 and the attribute column `attr` is equal to “A” .

```
RangeQuery Criterion query ;
query . mutableTable () = kTableName ;
query . mutableMaxVersions (). reset ( 1 );
{
    PrimaryKey & start = query . mutableInclusiveStart ();
    start . append () = PrimaryKey Column (
        " pkey ",
        PrimaryKey Value :: toInfMin ());
}
{
    PrimaryKey & end = query . mutableExclusiveEnd ();
    end . append () = PrimaryKey Column (
        " pkey ",
        PrimaryKey Value :: toInfMax ());
}
{
    // set filter
    shared_ptr < ColumnCondition > pkeyCond (
        new SingleColumnCondition (
            " pkey ",
            SingleColumnCondition :: kLarger ,
            AttributeValue :: toInteger ( 1 )));
```

```

shared_ptr < ColumnCondition > attrCond (
    new SingleColumnCondition (
        " attr ",
        SingleColumnCondition :: kEqual ,
        AttributeValue :: toString (" A "));
shared_ptr < CompositeColumnCondition > top ( new
CompositeColumnCondition ());
top -> mutableOp () = CompositeColumnCondition :: kAnd ;
top -> mutableChildren (). append () = pkeyCond ;
top -> mutableChildren (). append () = attrCond ;
query . mutableFilter () = top ;
}

```

7.8 Asynchronous interface

Create a client

- Create directly (use a Table Store endpoint to create a client)

```

Endpoint ep (" YourEndpoint ", " YourInstance ");
Credential cr (" AccessKeyId ", " AccessKeySecret ");
ClientOptions opts ;
AsyncClient * client = NULL ;
Optional < OTSError > res = AsyncClient :: create ( client ,
ep , cr , opts );

```



Notice:

Avoid using the AccessKey of your primary account to access Table Store. We recommend using a temporary token or the AccessKey of a sub-account. If you use a temporary STS token, the credential object in the aforementioned code must be changed to: `Credential cr (" AccessKeyId ", " AccessKeySecret ", " SecurityToken ");`.

See [Synchronous interfaces](#) for more information on the configuration items.

- Construct from SyncClient

```

SyncClient & sync = ... ;
AsyncClient * async = AsyncClient :: create ( sync );

```

Table operations

A table operation is used as an example here to demonstrate how to use an asynchronous interface.

Preparations

Two functions must be ready:

- Request object

The function signature of `listTable` is as follows:

```
void listTable (
    ListTableR equest &,
    const std::tr1::function< void (
        ListTableR equest &, util::Optional< OTSError >&,
        ListTableR response &)>&);
```

The first parameter is a mutable reference (in contrast, a synchronous interface is an immutable reference). After `listTable ()` is returned (at this point, the list table operation has not yet been completed), the passed-in `ListTableR equest` object may be changed or destructed, opening the door for subtle errors that are difficult to investigate. To avoid issues of this sort, asynchronous clients transfer (not copy) the contents of the passed-in request object to internal storage. This means that after calling `listTable ()`, passed-in objects can be changed.

- Callback function

The callback function does not return any value. The three types of receive parameters are as follows:

- Request object. This is the request object that is passed in when the user calls `listTable ()`. After callback, the asynchronous client no longer needs the request object and it is returned to the user's callback function in the form of a mutable reference. This allows the user to transfer out the contents of the request object.
- Error object wrapped in optional. If no errors exist, the `present ()` method of this object returns false.
- Response object. As with request objects, response objects are also returned to the callback function in the form of a mutable reference. If an error exists, the response object must be a valid object (one that can be destructed). However, the contents of the response object are undefined.



Note:

- The asynchronous client makes sure that every request callback is only called one time.
- Theoretically, a callback function can be called before the `listTable ()` is returned.

Example

```
void listTableC allback (
    ListTableR equest &,
    Optional < OTSError >& err ,
    ListTableR esponse & resp )
{
    if ( err . present () ) {
        // Error in processing
    } else {
        const IVector < string >& xs = resp . tables ();
        for ( int64_t i = 0 ; i < xs . size () ; ++ i ) {
            cout << xs [ i ] << endl ;
        }
    }
}

void listTable ( AsyncClient & client )
{
    ListTableR equest req ;
    client . listTable ( req , listTableC allback );
}
```

7.9 Logs

The client provides a logger by default. If your application has its own logger, we recommend using it, as this facilitates log management.

A logger has the following four elements, as defined in `tablestore / util / logger . hpp`.

- **Logger interface**

The Logger is used to assemble the contents of a log into a Record object that is forwarded to and written out by the Sinkers. The client also organizes the Logger into a tree structure, the root of which is the logger defined by the user in ClientOptions. The request logic and the networking logic use different sub-loggers derived from this root logger.

```
class Logger
{
public:
    enum LogLevel
    {
        kDebug ,
        kInfo ,
        kError ,
    };

    virtual ~ Logger () {}
    virtual LogLevel level () const = 0 ;
    virtual void record ( LogLevel , const std :: string &) = 0 ;
    virtual Logger * spawn ( const std :: string & key ) = 0 ;
};
```

```
virtual Logger * spawn ( const std::string & key ,
LogLevel ) = 0 ;
};
```

- `level ()` returns the log grade accepted by the Logger. Logs of lower grades are not transferred to the Sinkers.
- `record ()` accepts a log and its grade, and uses them to form a Record object that is sent to the corresponding Sinkers of the Logger.
- `spawn ()` derives a sub-logger.
- **Record interface**

Logger uses the Record object to transfer the contents of the log to the Sinkers.

The Record interface does not provide a method itself. The Logger and Sinkers decide which methods the Record class provides.

- **Sinkers interface**

Sinkers is tasked with writing out Record objects.

```
class Sinkers
{
public :
virtual ~ Sinkers () {}
virtual void sink ( Record *) = 0 ;
virtual void flush () = 0 ;
};
```

- `sink ()` writes out a Record. The Record can only be written to a cache.
- `flush ()` clears the cache to make sure that every log is stored.
- **SinkersCenter singleton object**

SinkersCenter holds all Sinkers objects and associates them with certain keys.

```
class SinkersCenter
{
public :
virtual ~ SinkersCenter () {}
static std::tr1::shared_ptr < SinkersCenter > singleton ();

virtual Sinkers * registerSinkers ( const std::string & key ,
Sinkers *) = 0 ;
virtual void flushAll () = 0 ;
};
```

- `singleton ()` obtains SinkersCenter singleton objects.
- `registerSinkers ()` registers a Sinkers in the SinkersCenter.
- `flushAll ()` clears all Sinkers in the SinkersCenter.

7.10 Error handling

TableStore C++ SDK uses return value to handle errors. All interfaces that can generate errors return `Optional < OTSError >` objects.

- `Optional < T >` is a template class defined in `tablestore/util/optional.hpp`. Think of it as a box that can store one `T` object at most. Inside the box are two scenarios that can be used to judge whether an error has occurred:
 - If a `T` object exists, it can be taken out and used. In this case, `Optional < T >::present ()` returns true, indicating that an error has occurred.
 - If no `T` object exists, `Optional < T >::present ()` returns false, indicating that no error occurred.
- The `OTSError` object indicates a specific error. It has 5 fields:
 - `httpStatus` and `errorCode`: HTTP return code and error code. In addition to the [Error messages](#), the following errors only happen on the client.

6	OTSCouldntResolveHost	Unable to resolve the domain name. The instance access address is wrong or the network is disconnected.
7	OTSCouldntConnect	Unable to connect to the service. Configuration error in local host files.
28	OTSRequestTimeout	Request timeout.
35	OTSSslHandshakeFail	HTTPS handshake failure. Local certificate not installed.
55	OTSWriteRequestFail	Network delivery failure . Network interruption.
56	OTSCorruptedResponse	Incomplete response.

89	OTSTNoAvailableConnection	No available connection . This usually happens in newly constructed clients or when concurrent requests exceed the total number of network connections.
----	---------------------------	---

- `message` : Error details.
- `requestId` : Each request sent to the service is assigned a number by the service. If a response is returned normally, the response object contains the `requestId`. If the service determines that a request has errors, the error object contains the `requestId`. If errors occur before the request is sent, or if they happen on the network link, the error object does not contain the `requestId`.
- `traceId` : Each API call is assigned a `traceId` by the client. Different API calls are assigned a different `traceId`. If the same API call is tried repeatedly, the `traceId` remains the same but the `requestId` may vary.

7.11 Retry

Common retry strategies

The SDK provides the following common retry strategies:

- Default retry strategy

An SDK error automatically triggers retries. The maximum retry interval is 10 seconds.

- Counting retry strategy

Retry at user-defined intervals. If the retry strategy is secure, the maximum number of retry attempts cannot exceed the user-defined number.

- No retry

Custom retry strategy

You can customize your retry strategy by modifying `RetryStrategy`.

```
class RetryStrategy
{
public:
    virtual ~RetryStrategy () {}

    virtual RetryStrategy * clone () const = 0 ;
    virtual int64_t retries () const throw () = 0 ;
};
```

```

    virtual bool shouldRetry ( Action , const OTSError &)
    const = 0 ;
    virtual util :: Duration nextPause () = 0 ;
};

```

- `clone ()` replicates a new object. The new object must be of the same type and internal status (including the number of retries) as the current one.
- `retries ()`, number of already completed retries.
- `shouldRetry ()` specifies operations and errors to determine whether a retry is needed.

Table Store provides two tool functions for easier operation:

- The first retrieable divides errors into three groups.
 - Completely harmless retries, such as OTSTableNotReady.
 - Harmful or meaningless retries, such as with parameter errors.
 - Cannot be judged based on errors alone, such as OTSRequestTimeout.
- The second retrieable determines whether a retry can be performed based on the idempotence principles of an operation, while also taking operations and errors into consideration. In other words, both RETRIABLE errors and the DEPENDS errors of a read operation are judged to be retrieable.

```

enum RetryCategory
{
    UNRETRIABLE ,
    RETRIABLE ,
    DEPENDS ,
};
static RetryCategory retrieable ( const OTSError &);
static bool retrieable ( Action , const OTSError &);

```

`nextPause ()` is the interval until the next retry, if retrieable.

8 PHP SDK

8.1 Overview

Description

This topic describes how to install and use PHP SDK V4.0.0 and later for Table Store. This topic assumes that you have activated Table Store and created an AccessKey ID and an AccessKey Secret.

- If you have not activated Table Store or want to learn more about Table Store, visit the [Table Store product page](#).
- If you do not have an AccessKey ID and an AccessKey Secret, create an AccessKey in the [Alibaba Cloud console](#).

Note

SDK V4.0.0 and later support multiple data versions and time to live (TTL). However, these SDK versions are not compatible with SDK V2.x.x.

- [TTL](#)
- [Multiple data versions](#)
- [Auto-increment primary key column](#)

SDK download

- [SDK source code package](#)
- [GitHub](#)

For more information about version iterations, see [PHP SDK version history](#).

Version

Latest version: 5.0.0

Compatibility

For SDKs V4.x.x:

- Compatible

For SDKs V2.x.x:

- Incompatible

Change

- 5.5.0

- New features: Local transactions are supported.
- New features: Search indexes are supported.
- New features: Secondary indexes are supported.

- 4.1.0

Basic Stream operations are supported.

- 4.0.0

- PHP 5.5 and later are supported, such as 5.5, 5.6, 7.0, 7.1, and 7.2. Only 64-bit PHP systems are supported, and PHP 7 is recommended.
- New features: TTL setting is enabled, and the `table_options` parameter is added to `CreateTable` and `UpdateTable`.
- New features: Multiple data versions are supported. The timestamp setting is enabled for `PutRow`, `UpdateRow`, `DeleteRow`, and `BatchGetRow`. The `max_versions` parameter is added to `GetRow`, `GetRange`, and `BatchGet`.
- New features: The auto increment feature is enabled for the primary key column. The `return_type` parameter is added to operations. The `primary_key` parameter is added to the response to return the primary key of the corresponding operation.
- Changes: The underlying protobuf library is upgraded to the Google protobuf-php library.
- Changes: The value of the `primary_key` parameter for each operation is changed to the list type to ensure the order of primary key columns.
- Changes: The value of the `attribute_columns` parameter for each operation is changed to the list type to support multiple data versions.

8.2 Installation

Environment preparation

- 64-bit PHP 5.5 or later (required)

You can run the `php -v` command to check the current PHP version. Table Store uses 64-bit integers. However, only the string type can be used to represent the 64-bit integer type in 32-bit PHP. Therefore, Table Store does not support 32-bit

PHP. In Windows, PHP versions earlier than PHP 7 are not actually 64-bit. When running PHP in Windows, upgrade PHP to PHP 7 or modify the environment by yourself. We strongly recommend that you use PHP 7 to obtain the optimal performance.

- **cURL extension (recommended):** You can run the `php -m` command to check whether cURL extension is installed.



Note:

- In Ubuntu, you can run the `sudo apt-get install php-curl` command to install cURL extension of PHP through apt-get.
 - In CentOS, you can run the `sudo yum install php-curl` command to install cURL extension of PHP through yum.
- **OpenSSL extension (recommended):** To use HTTPS, install OpenSSL PHP extension
 -

Installation methods

Composer

To install the SDK through Composer, follow these steps:

1. Run the `composer require aliyun / aliyun - tablestore - sdk - php` command in the root directory of the project or declare the dependency on Table Store PHP SDK in the `composer.json` file.

```
"require": {  
    "aliyun / aliyun - tablestore - sdk - php": "~ 5 . 0 "  
}
```

2. Run the `composer install` command to install the dependency. After the dependency is installed, check whether your directory structure complies with the following structure:

```
i  
├── app . php  
├── composer . json  
├── composer . lock  
└── vendor
```

In the preceding directory structure, `app.php` indicates your application. The `vendor` directory contains the dependent library. You must introduce the dependency into `app.php`.

```
require_once __DIR__ . '/vendor/autoload.php';
```



Note:

- If your project has referenced `autoload.php`, you do not need to introduce it again after adding the SDK dependency.
- If a network error occurs when you use Composer, you can use the [image source](#) of Composer China by running the `composer config -g repositories .packagist composer http://packagist.phpcomposer.com` command.

Source code package

You can download the source code package in either of the following ways:

- Select the required version and download the ZIP package on [GitHub](#).
- Click [here](#) to download the source code package.

Sample programs

The Table Store PHP SDK provides a variety of sample programs for your reference or use. You can obtain the sample programs by using either of the following methods:

- Download and decompress the Table Store PHP SDK package, and locate the sample programs in the `examples` directory.
- Access the GitHub project for the Table Store PHP SDK at [aliyun-tablestore-php-sdk](#).

To run a sample program, follow these steps:

- Decompress the downloaded SDK package.
- Modify the `ExampleConfig.php` file in the `examples` directory.

```
EXAMPLE_ENDPOINT : the AccessKey ID that you obtained
from Alibaba Cloud .
EXAMPLE_ACCESS_KEY_ID : the AccessKey Secret that you
obtained from Alibaba Cloud .
```

```
EXAMPLE_ACCESS_KEY_SECRET : the domain name that you
use to access the Table Store data center , for
example , https://sun.cn-hangzhou.ots.aliyuncs.com.
EXAMPLE_INSTANCE_NAME : the instance that you use
to run the sample program . The sample program is
operated in this instance .
```

- Run a sample file in the examples directory separately.

The following table describes the sample files.

File	Description
NewClient.php	Demonstrates how to set the default client.
NewClient2.php	Demonstrates how to customize the client.
NewClientLogClosed.php	Demonstrates how to disable logging on the client.
NewClientLogDefined.php	Demonstrates how to customize logging on the client.
CreateTable.php	Demonstrates how to use CreateTable.
DeleteTable.php	Demonstrates how to use DeleteTable.
DescribeTable.php	Demonstrates how to use DescribeTable.
ListTable.php	Demonstrates how to use ListTable.
UpdateTable.php	Demonstrates how to use UpdateTable.
ComputeSplitPointsBySize.php	Demonstrates how to use ComputeSplitPointsBySize.
PutRow.php	Demonstrates how to use PutRow.
PutRowWithColumnFilter.php	Demonstrates how to use PutRow with conditional update.
UpdateRow1.php	Demonstrates how to use PUT in UpdateRow.
UpdateRow2.php	Demonstrates how to use DELETE_ALL in UpdateRow.
UpdateRow3.php	Demonstrates how to use DELETE in UpdateRow.
UpdateRowWithColumnFilter.php	Demonstrates how to use UpdateRow with conditional update.
GetRow.php	Demonstrates how to use GetRow.

File	Description
GetRow2.php	Demonstrates how to set column_to_get in GetRow.
GetRowWithSingleColumnFilter.php	Demonstrates how to use GetRow with conditional filtering.
GetRowWithMultipleColumnFilter.php	Demonstrates how to use GetRow with complex conditional filtering.
DeleteRow.php	Demonstrates how to use DeleteRow.
DeleteRowWithColumnFilter.php	Demonstrates how to use DeleteRow with conditional filtering.
PKAutoIncrment.php	Demonstrates how to use auto-increment columns in detail.
BatchGetRow1.php	Demonstrates how to use BatchGetRow to read multiple rows from a table.
BatchGetRow2.php	Demonstrates how to use BatchGetRow to read multiple rows from multiple tables.
BatchGetRow3.php	Demonstrates how to use BatchGetRow to read multiple rows from a table with the specified columns.
BatchGetRow4.php	Demonstrates how to use BatchGetRow to process returned results.
BatchGetRowWithColumnFilter.php	Demonstrates how to use BatchGetRow with conditional filtering.
BatchWriteRow1.php	Demonstrates how to perform multiple PUT operations in BatchWriteRow.
BatchWriteRow2.php	Demonstrates how to perform multiple UPDATE operations in BatchWriteRow.
BatchWriteRow3.php	Demonstrates how to perform multiple DELETE operations in BatchWriteRow.
BatchWriteRow4.php	Demonstrates how to simultaneously perform the UPDATE, PUT, and DELETE operations in BatchWriteRow.
BatchWriteRowWithColumnFilter.php	Demonstrates how to use BatchWriteRow with conditional update.
GetRange1.php	Demonstrates how to use GetRange.

File	Description
GetRange2.php	Demonstrates how to use GetRange to obtain the specified columns.
GetRange3.php	Demonstrates how to use GetRange to obtain the specified number of rows.
GetRangeWithColumnFilter.php	Demonstrates how to use GetRange with conditional filtering.

8.3 Initialization

The OTSClient is the client for Table Store. It provides a series of methods for callers to perform operations on tables and read data from or write data to a single row or multiple rows. To use the PHP SDK to initiate a request to Table Store, you need to initialize an OTSClient instance and modify the default configurations of OTSClientC onfig as required.

- You can access a Table Store instance over the Internet or intranet.
- You can log on to the [Table Store console](#) and go to the instance details page. The instance access URL on the page is the endpoint for that instance.

Determine an endpoint

An endpoint is the domain of a Table Store instance in a region. It supports the formats listed in the following table.

Example	Description
<code>http://sun.cn-hangzhou.ots.aliyuncs.com</code>	Accesses the sun instance in Hangzhou over the Internet by using the HTTP protocol.
<code>https://sun.cn-hangzhou.ots.aliyuncs.com</code>	Accesses the sun instance in Hangzhou over the Internet by using the HTTPS protocol.



Note:

- You can access a Table Store instance over the Internet or intranet.
- You can log on to the [Table Store console](#) and go to the instance details page. The instance access URL on the page is the endpoint for that instance.

Configure an AccessKey

To access Table Store, you must have a valid AccessKey for signature authentication. The following types of AccessKeys are supported:

- To obtain the AccessKey ID and AccessKey Secret of the Alibaba Cloud account, follow these steps:
 1. Register an [Alibaba Cloud account](#) on the Alibaba Cloud website.
 2. Log on to the Alibaba Cloud console and [apply for an AccessKey](#).
- To obtain the AccessKey ID and AccessKey Secret of a RAM user authorized to access Table Store, follow these steps:
 1. Use your Alibaba Cloud account to access [RAM](#) and create a RAM user or use an existing one.
 2. Use your Alibaba Cloud account to authorize the RAM user to access Table Store.
 3. After authorization, the AccessKey ID and AccessKey Secret of the RAM user can be used to access Table Store.
- To obtain an STS token for temporary access, follow these steps:
 1. Obtain the temporary AccessKey ID, AccessKey Secret, and token from the application server, which accesses RAM or STS to obtain the preceding information.
 2. Use the received AccessKey ID, AccessKey Secret, and token to access Table Store.

Initialize an OTSClient instance

After obtaining the AccessKey ID and AccessKey Secret, initialize an OTSClient instance.

1. Use the endpoint of Table Store to create a client. An example is as follows:

```
$ otsClient = new OTSClient ( array (
    ' EndPoint ' => "< your endpoint >",
    ' AccessKeyI D ' => "< your access id >",
    ' AccessKeyS ecret ' => "< your access key >"
    ' InstanceNa me ' => "< your instance name >"
)
```

```
));
```

Note:

- In addition to the AccessKey ID and AccessKey Secret, the STS token can also be used to access Table Store.
- When creating an OTSClient instance, you can set parameters such as the timeout time, the maximum number of connections, and the maximum number of retries in OTSClientConfig.

2. Configure the OTSClient instance.

To modify the default configuration of the OTSClient instance, import the corresponding parameters, such as the proxy, connection timeout time, and the maximum number of connections, when constructing the OTSClient instance. The following table describes the parameters to be set.

Parameter	Description	Default value
ConnectionTimeout	The maximum latency allowed to connect to Table Store.	2.0 seconds
StsToken	The token for temporary access.	null
SocketTimeout	The maximum latency allowed for the response to each request.	2.0 seconds. We recommend that you set this parameter to a larger value when transmitting a large volume of data.
RetryPolicy	The retry policy.	DefaultRetryPolicy. If the value is null, the retry feature is disabled.
ErrorLogHandler	The error-level log processing function, which is used to display the logs of errors returned by Table Store.	defaultOTSErrorLogHandler. If the value is null, this function is disabled.
DebugLogHandler	The debug-level log processing function, which is used to display the logs of normal requests and responses.	defaultOTSDebugLogHandler. If the value is function, this feature is disabled.

HTTPS

Install the OpenSSL PHP extension.

8.4 Table operations

8.4.1 Create a table

CreateTable

API description: [CreateTable](#)

You can call the CreateTable operation to create a table. When creating a table in Table Store, you must specify TableMeta and TableOptions. You can also set ReservedThroughput.

After you create the table, the server loads partitions of the table to specified nodes. Therefore, you must wait several seconds before reading data from or writing data to the table. Otherwise, the system throws an exception. Note: If the system throws an exception, the SDK automatically retries the read or write operation.

Method

```
/**
 * Create a table . Set the number , names , sequence
 * , and types of primary key columns , reserved read
 * and write throughput , TTL , and Stream options .
 * @ api
 * @ param [] $ request The request parameters .
 * @ return [] The response . If the CreateTable e
 * operation succeeds , no message is returned . An empty
 * array is returned here to be consistent with other
 * operations .
 * @ throws OTSClientE xception The exception that is
 * thrown when a parameter check error occurs or the
 * Table Store server returns a verificati on error .
 * @ throws OTSServerE xception The exception that is
 * thrown when the Table Store server returns an error .
 */
public function createTable ( array $ request );
```

Request format

```
$ result = $ client -> createTable ([
    ' table_meta ' => [ // Required .
        ' table_name ' => '< string >',
        ' primary_key_schema ' => [
            ['< string >', < PrimaryKey Type >],
            ['< string >', < PrimaryKey Type >],
            ['< string >', < PrimaryKey Type >, < PrimaryKey Option
        ]
    ]
]);
```

```

    ],
    'reserved_throughput' => [           // Required .
        'capacity_unit' => [
            'read' => < integer >,
            'write' => < integer >
        ]
    ],
    'table_options' => [                 // Required .
        'time_to_live' => < integer >,
        'max_versions' => < integer >,
        'deviation_cell_version_in_sec' => < integer >
    ],
    'stream_spec' => [
        'enable_stream' => true || false ,
        'expiration_time' => < integer >
    ]
]);

```

Request format description

- **table_meta:** required. The table name and the schema of the primary key.
 - **table_name:** the table name.
 - **primary_key_schema:** the schema of the primary key of the table.
- The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key schema PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
- The first primary key column serves as the partition key. Data with the same partition key value is in the same partition. Therefore, we recommend that you assign no more than 10 GB of data with the same partition key value. Otherwise, individual partitions are too large and cannot be split. We also recommend that you distribute read and write operations evenly among different partitions to facilitate load balancing.
- When creating a table, you only need to define the primary key of the table. Attribute columns do not need to be defined. Different rows can have different attribute columns and the names of attribute columns can be specified when data is written. Therefore, Table Store is suitable for the

storage of semi-structured data. You can dynamically add new attribute columns during business development.

- Each item consists of the primary key name, primary key type (`PrimaryKeyType`), and primary key settings (`PrimaryKeyOption`, which is optional).
 - `PrimaryKeyType` can be set to `PrimaryKeyTypeConst::CONST_INTEGER`, `PrimaryKeyTypeConst::CONST_STRING`, or `PrimaryKeyTypeConst::CONST_BINARY`, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), and BINARY types.
 - `PrimaryKeyOption` can be set to `PrimaryKeyOptionConst::CONST_PK_AUTO_INCR`, which indicates the auto-increment column. For more information, see the description about auto-increment primary key columns.
- `reserved_throughput`: required. The reserved read throughput and reserved write throughput, which are related to billing.
 - `capacity_unit`: the reserved read throughput and reserved write throughput. When the reserved read throughput or reserved write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds the reserved volume is billed on a pay-as-you-go basis. By default, both the reserved read throughput and reserved write throughput are 0. That is, all traffic is billed on a pay-as-you-go basis by default. To set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can set the reserved read throughput and reserved write throughput of capacity instances only to 0. These instances do not allow throughput reservation.
 - `read`: the reserved read throughput.
 - `write`: the reserved write throughput.
 - `table_options`: required. The TTL, MaxVersions, and MaxTimeDeviation configurations of the table.
 - `time_to_live`: the retention time of data. Unit: seconds.
 - The latest version of Table Store API supports automatic data expiration. If you do not want data to expire, set `time_to_live` to -1.
 - Table Store checks whether data has expired based on the data timestamp, current time, and table TTL. If the difference between the current time and the data timestamp is greater than the table TTL, data expires and is cleared

by the Table Store server. For more information about the data timestamp, see [Preface](#).

- After the TTL is set, the server checks whether data expires based on the data timestamp. If you specify a timestamp when writing data and the timestamp is significantly different from the actual time, unexpected data expiration may occur. For example, if the specified data timestamp is too small, the data may expire and be deleted as soon as it is written. If the specified timestamp is too large, the data may not expire when required. Therefore, when writing data with the TTL specified, make sure that you set a reasonable timestamp.
- `max_versions`: the maximum number of versions retained by each attribute column.
 - The latest version of Table Store API supports data models of multiple versions. For more information, see [Preface](#). The `max_versions` parameter is used to specify the maximum number of data versions saved by each attribute column. If a new version causes the total number of versions to exceed the value of `max_versions`, the server deletes the earliest version to ensure that the number of versions do not exceed this value.
- `deviation_cell_version_in_sec`: the maximum deviation allowed between the specified version and the current system time when data is written to the version. Unit: seconds.
 - Table Store supports multiple versions. By default, the system generates a new version number when writing new data. The version number is a timestamp corresponding to the write time in milliseconds. The TTL feature uses this timestamp to determine if data has expired. In addition, you can specify the data writing timestamp. If you set an extremely small timestamp, which deviates from the current time by more than the TTL set for the table, the written data immediately expires. To prevent immediate data expiration, Table Store provides the `deviation_cell_version_in_sec` parameter to limit the permissible deviation between the data writing timestamp and the current system time. The value of this parameter is expressed in seconds. You can specify this parameter when creating a table and modify it by using the `UpdateTable` operation.

- **stream_spec**: optional. The Stream-related settings.
 - **enable_stream**: specifies whether to enable Stream.
 - **expiration_time**: the expiration time of the Stream data of the table. Unit: hours. Earlier modification records are deleted.

Response format

```
[ ]
```

Response format description

The response is empty. If an error occurs, the system throws an exception.

Example

The following code provides an example of how to create a table with three primary key columns, set the reserved read and write throughput to 0 and TTL to -1, and then enable Stream.

```
// Create a schema for the primary key columns
// including the number, names, and types of the
// primary key columns.
// The first primary key column (partition column)
// is named PK0 and belongs to the integer type.
// The second primary key column is named PK1
// and belongs to the string type.
// The third primary key column is named PK2
// and belongs to the binary type.
$result = $client->createTable([
    'table_meta' => [
        'table_name' => 'SampleTable',
        'primary_key_schema' => [
            ['PK0', PrimaryKeyTypeConst::CONST_INTEGER],
            ['PK1', PrimaryKeyTypeConst::CONST_STRING],
            ['PK2', PrimaryKeyTypeConst::CONST_BINARY]
        ]
    ],
    'reserved_throughput' => [
        'capacity_unit' => [
            'read' => 0,
            'write' => 0
        ]
    ],
    'table_options' => [
        'time_to_live' => -1,
        'max_versions' => 2,
        'deviation_cell_version_in_sec' => 86400
    ],
    'stream_spec' => [
        'enable_stream' => true,
        'expiration_time' => 24
    ]
]);
```

```
]);
```

8.4.2 Obtain table names

ListTable

API description: [ListTable](#)

You can call the ListTable operation to obtain the names of all tables created in the current instance.

Method

```
/**
 * Obtain the names of all tables created in the
 * current instance .
 * @ api
 * @ param [] $ request The request parameters , which
 * are empty .
 * @ return [] The response .
 * @ throws OTSClientE xception The exception that is
 * thrown when a parameter check error occurs or the
 * Table Store server returns a verificati on error .
 * @ throws OTSServerE xception The exception that is
 * thrown when the Table Store server returns an error .
 */
public function listTable ( array $ request );
```

Request format

```
$ result = $ client -> listTable ( [] );
```

Request format description

Currently, no parameter is required in the request.

Response format

```
[
    '< string >',
    '< string >',
    '< string >'
]
```

Response format description

The result is a list of strings. Each item indicates a table name.

Example

The following code provides an example of how to obtain the names of all tables in an instance:

```
$ result = $ otsClient -> listTable ([]);
```

8.4.3 Update a table

UpdateTable

API description: [UpdateTable](#)

You can call the UpdateTable operation to update ReservedThroughput, TableOptions, and StreamSpecification of a table.

For more information about ReservedThroughput, TableOptions, and StreamSpecification, see the topic about how to create a table. Currently, the interval between attempts to modify the value of ReservedThroughput is a minimum of 1 minute.

Method

```
/**
 * Update a table , including the reserved read
 * throughput and reserved write throughput , configurat ion
 * informatio n , and Stream configurat ion of the table .
 * This method can be used to increase or
 * decrease the reserved read throughput and reserved
 * write throughput .
 * @ api
 * @ param [] $ request The request parameters .
 * @ return [] The response .
 * @ throws OTSClientE xception The exception that is
 * thrown when a parameter check error occurs or the
 * Table Store server returns a verificati on error .
 * @ throws OTSServerE xception The exception that is
 * thrown when the Table Store server returns an error .
 */
public function updateTabl e ( array $ request );
```

Request format

```
$ result = $ client -> updateTabl e ([
    ' table_name ' => '< string >',           // Required .
    ' reserved_t hroughput ' => [
        ' capacity_u nit ' => [
            ' read ' => < integer >,
            ' write ' => < integer >
        ]
    ],
    ' table_opti ons ' => [
        ' time_to_li ve ' => < integer >,
        ' max_versio ns ' => < integer >,
        ' deviation_ cell_versi on_in_sec ' => < integer >
    ],
    ' stream_spe c ' => [
```

```

        'enable_streams' => true || false ,
        'expiration_time' => < integer >
    ];
});

```

Request format description

- The difference between UpdateTable and CreateTable is the TableMeta parameter. Except TableMeta, all other parameters are defined and set in the same way as those of CreateTable. Except table_name, all other parameters are optional.
- table_name: required. The table name.
- reserved_throughput: optional. The reserved read throughput and reserved write throughput, which are related to billing.
 - capacity_unit: the reserved read throughput and reserved write throughput. When the reserved read throughput or reserved write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds the reserved volume is billed on a pay-as-you-go basis. By default, both the reserved read throughput and reserved write throughput are 0. That is, all traffic is billed on a pay-as-you-go basis by default. To set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can set the reserved read throughput and reserved write throughput of capacity instances only to 0. These instances do not allow throughput reservation.
 - read: the reserved read throughput.
 - write: the reserved write throughput.
- table_options: optional. The TTL, MaxVersions, and MaxTimeDeviation configurations of the table.
 - time_to_live: the retention time of data. Unit: seconds.
 - The latest version of Table Store API supports automatic data expiration. If you do not want data to expire, set time_to_live to -1.
 - Table Store checks whether data has expired based on the data timestamp, current time, and table TTL. If the difference between the current time and the data timestamp is greater than the table TTL, data expires and is cleared

by the Table Store server. For more information about the data timestamp, see [Preface](#).

- After the TTL is set, the server checks whether data expires based on the data timestamp. If you specify a timestamp when writing data and the timestamp is significantly different from the actual time, unexpected data expiration may occur. For example, if the specified data timestamp is too small, the data may expire and be deleted as soon as it is written. If the specified timestamp is too large, the data may not expire when required. Therefore, when writing data with the TTL specified, make sure that you set a reasonable timestamp.
- `max_versions`: the maximum number of versions retained by each attribute column.
 - The latest version of Table Store API supports data models of multiple versions. For more information, see [Preface](#). The `max_versions` parameter is used to specify the maximum number of data versions saved by each attribute column. If a new version causes the total number of versions to exceed the value of `max_versions`, the server deletes the earliest version to ensure that the number of versions do not exceed this value.
- `deviation_cell_version_in_sec`: the maximum deviation allowed between the specified version and the current system time when data is written to the version. Unit: seconds.
 - Table Store supports multiple versions. By default, the system generates a new version number when writing new data. The version number is a timestamp corresponding to the write time in milliseconds. The TTL feature uses this timestamp to determine if data has expired. In addition, you can specify the data writing timestamp. If you set an extremely small timestamp, which deviates from the current time by more than the TTL set for the table, the written data immediately expires. To prevent immediate data expiration, Table Store provides the `deviation_cell_version_in_sec` parameter to limit the permissible deviation between the data writing timestamp and the current system time. The value of this parameter is expressed in seconds. You can specify this parameter when creating a table and modify it by using the `UpdateTable` operation.

- **stream_spec**: optional. The Stream-related settings.
 - **enable_stream**: specifies whether to enable Stream.
 - **expiration_time**: the expiration time of the Stream data of the table. Unit: hours. Earlier modification records are deleted.

Response format

```
[
  'capacity_unit_details' => [
    'capacity_unit' => [
      'read' => < integer >,
      'write' => < integer >
    ],
    'last_increase_time' => < integer >,
    'last_decrease_time' => < integer >
  ],
  'table_options' => [
    'time_to_live' => < integer >,
    'max_versions' => < integer >,
    'deviation_cell_version_in_sec' => < integer >
  ],
  'stream_details' => [
    'enable_stream' => true || false ,
    'stream_id' => '< string >',
    'expiration_time' => < integer >,
    'last_enable_time' => < integer >
  ]
]
```

Response format description

- **capacity_unit_details**: the reserved read throughput and reserved write throughput, which are related to billing.
 - **capacity_unit**: the reserved read throughput and reserved write throughput. When the reserved read throughput or reserved write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds the reserved volume is billed on a pay-as-you-go basis. By default, both the reserved read throughput and reserved write throughput are 0. That is, all traffic is billed on a pay-as-you-go basis by default. To set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can set the reserved read throughput and reserved

write throughput of capacity instances only to 0. These instances do not allow throughput reservation.

■ read: the reserved read throughput.

■ write: the reserved write throughput.

- last_increase_time: the time when the reserved read throughput or reserved write throughput was increased most recently for this table. The time is expressed in the number of seconds that have elapsed since 00:00:00, January 1, 1970 (UTC).
- last_decrease_time: the time when the reserved read throughput or reserved write throughput was decreased most recently for this table. The time is expressed in the number of seconds that have elapsed since 00:00:00, January 1, 1970 (UTC).
- table_options: the TTL, MaxVersions, and MaxTimeDeviation configurations of the table. The options are the same as those in the request.
- stream_details: the Stream information of the table.
 - enable_stream: indicates whether Stream is enabled for the table.
 - stream_id: the Stream ID of the table.
 - expiration_time: the expiration time of the Stream data of the table. Unit: hours. Earlier modification records are deleted.
 - last_enable_time: the time when Stream was enabled most recently.

Example

The following code provides an example of how to update the numbers of read and write CUs of a table to 1 and 2, respectively:

```
$ result = $ client -> updateTable ([
    'table_name' => 'SampleTable',
    'reserved_throughput' => [
        'capacity_unit' => [
            'read' => 1, // The number of
read or write CUs can be separately updated.
            'write' => 2
        ]
    ]
]);
```

The following code provides an example of how to set the TTL of a table to 86,400s (one day), retain two versions, and set the maximum deviation to 10s:

```
$ result = $ client -> updateTable ([
```

```

        ' table_name ' => ' SampleTabl e ',
        ' table_opti ons ' => [
            ' time_to_li ve ' => 86400 ,
            ' max_versio ns ' => 2 ,
            ' deviation_ cell_versi on_in_sec ' => 10
        ]
    });

```

The following code provides an example of how to enable Stream for a table and set the expiration time to 24 hours:

```

    $ result = $ client -> updateTabl e ([
        ' table_name ' => ' SampleTabl e ',
        ' stream_spe c ' => [
            ' enable_str eam ' => true ,
            ' expiration _time ' => 24
        ]
    ]);

```

8.4.4 Delete a table

DeleteTable

API description: [DeleteTable](#)

You can call the DeleteTable operation to delete a specified table from the current instance.

Method

```

/**
 * Delete a table based on the table name .
 * @ api
 * @ param [] $ request The request parameters .
 * @ return [] The response . If the DeleteTabl e
operation succeeds , no message is returned . An empty
array is returned here to be consistent with other
operations .
 * @ throws OTSClientE xception The exception that is
thrown when a parameter check error occurs or the
Table Store server returns a verificati on error .
 * @ throws OTSServerE xception The exception that is
thrown when the Table Store server returns an error .
 */
public function deleteTabl e ( array $ request );

```

Request format

```

    $ result = $ client -> deleteTabl e ([
        ' table_name ' => '< string >', // Required .
    ]);

```

Request format description

table_name: the table name.

Response format

```
[]
```

Response format description

The response is empty. If an error occurs, the system throws an exception.

Example

The following code provides an example of how to delete a table:

```
$ result = $ otsClient -> deleteTable ([
    'table_name' => ' MyTable '
]);
```

8.4.5 Split data with a specified size

ComputeSplitsBySize

API description: [ComputeSplitPointsBySize](#)

You can call the **ComputeSplitsBySize** operation to logically split data in a table into several partitions whose sizes are close to the specified size. This operation returns the split points between partitions and prompt about hosts where the partitions reside. This operation is generally used for execution plans on computing engines, such as concurrency plans.

Method

```
/**
 * Logically split data in a table into several
 * partitions whose sizes are close to the specified
 * size , and return the split points between partitions
 * and prompt about hosts where the partitions reside .
 * This operation is generally used for execution
 * plans on computing engines , such as concurrency plans
 * .
 * @ api
 * @ param [] $ request The request parameters .
 * @ return [] The response .
 * @ throws OTSClientException The exception that is
 * thrown when a parameter check error occurs or the
 * Table Store server returns a verification error .
 * @ throws OTSServerException The exception that is
 * thrown when the Table Store server returns an error .
 */
public function computeSplitPointsBySize ( array $
request )
```

Request format

```
$ result = $ client -> ComputeSplitSizeBySize ([
    ' table_name ' => '< string >', // Required .
    ' split_size ' => < integer > // Required .
]);
```

Request format description

- **table_name**: the table name.
- **split_size**: the approximate size of each partition. Unit: megabytes.

Response format

```
[
    ' consumed ' => [
        ' capacity_unit ' => [
            ' read ' => < integer >,
            ' write ' => < integer >
        ]
    ],
    ' primary_key_schema ' => [
        ['< string >', < PrimaryKey Type >],
        ['< string >', < PrimaryKey Type >, < PrimaryKey Option >]
    ]
    ' splits ' => [
        [
            ' lower_bound ' => [
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >],
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >]
            ],
            ' upper_bound ' => [
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >],
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >]
            ],
            ' location ' => '< string >'
        ],
        // ...
    ]
]
```

Response format description

- **consumed**: the number of CUs consumed by this operation.
- **capacity_unit**: the number of read and write CUs consumed.
 - **read**: the number of read CUs consumed.
 - **write**: the number of write CUs consumed.

- **primary_key_schema**: the schema of the primary key of the table, which is the same as that specified when the table was created.
- **splits**: the split points between partitions.
 - **lower_bound**: the minimum value in the range of the primary key. Note: This parameter value can be used by `GetRange`.
 - Each item contains the primary key name, primary key value (`PrimaryKeyValue`), and primary key type (`PrimaryKeyType`).
 - `PrimaryKeyType` can be set to `PrimaryKeyTypeConst::CONST_INTEGER`, `PrimaryKeyTypeConst::CONST_STRING`, `PrimaryKeyTypeConst::CONST_BINARY`, `PrimaryKeyTypeConst::CONST_INF_MIN`, or `PrimaryKeyTypeConst::CONST_INF_MAX`, which respectively indicate the `INTEGER`, `STRING` (UTF-8 encoded string), `BINARY`, `INF_MIN(-inf)`, and `INF_MAX(inf)` types.
 - **upper_bound**: the maximum value in the range of the primary key. The format is the same as that of `lower_bound`.
 - **location**: the prompt about the hosts where the partitions reside. This parameter can be null.

Example

The following code provides an example of how to split data with a specified size:

```
$ result = $ client -> ComputeSplitsBySize ([
    ' table_name ' => ' MyTable ',
    ' split_size ' => 1
]);
foreach ($ result [' splits ' ] as $ split ) {
    print_r ($ split [' location ' ]);
    print_r ($ split [' lower_bound ' ]);
    print_r ($ split [' upper_bound ' ]);
}
```

8.4.6 Query the description of a table

DescribeTable

API description: [DescribeTable](#)

You can call the `DescribeTable` operation to query `TableMeta`, `TableOptions`, `ReservedThroughputDetails`, and `StreamDetails` of a table. `TableMeta` and `TableOptions` have already been described in the topic about how to create a table. Besides the reserved read throughput and reserved write throughput, `ReservedThroughputDe`

tails shows the time when the throughput was increased or decreased most recently.

StreamDetails shows the detailed information about Stream.

Method

```
/**
 * Obtain information about a table, including the
 * primary key schema, reserved read throughput, and
 * reserved write throughput.
 * @api
 * @param [] $request The request parameters.
 * @return [] The response.
 * @throws OTSClientException The exception that is
 * thrown when a parameter check error occurs or the
 * Table Store server returns a verification error.
 * @throws OTSServerException The exception that is
 * thrown when the Table Store server returns an error.
 */
public function describeTable ( array $request );
```

Request format

```
$result = $client->describeTable ([
    'table_name' => '<string>', // Required.
]);
```

Request format description

- **table_name:** the table name.

Response format

```
[
    'table_meta' => [
        'table_name' => '<string>',
        'primary_key_schema' => [
            '<string>', <PrimaryKey Type>,
            '<string>', <PrimaryKey Type>, <PrimaryKey Option>
        ]
    ],
    'capacity_unit_details' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ],
        'last_increase_time' => <integer>,
        'last_decrease_time' => <integer>
    ],
    'table_options' => [
        'time_to_live' => <integer>,
        'max_versions' => <integer>,
        'deviation_cell_version_in_sec' => <integer>
    ],
    'stream_details' => [
        'enable_stream' => true || false,
        'stream_id' => '<string>',
        'expiration_time' => <integer>,
    ]
]
```

```

        'last_enabled_time' => < integer >
    ]
]

```

Response format description

- **table_meta**: the table name and the schema of the primary key, which are the same as those specified when the table was created.
- **capacity_unit_details**: the reserved read throughput and reserved write throughput, which are related to billing.
 - **capacity_unit**: the reserved read throughput and reserved write throughput. When the reserved read throughput or reserved write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds the reserved volume is billed on a pay-as-you-go basis. By default, both the reserved read throughput and reserved write throughput are 0. That is, all traffic is billed on a pay-as-you-go basis by default. To set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can set the reserved read throughput and reserved write throughput of capacity instances only to 0. These instances do not allow throughput reservation.
 - **read**: the reserved read throughput.
 - **write**: the reserved write throughput.
 - **last_increase_time**: the time when the reserved read throughput or reserved write throughput was increased most recently for this table. The time is expressed in the number of seconds that have elapsed since 00:00:00, January 1, 1970 (UTC).
 - **last_decrease_time**: the time when the reserved read throughput or reserved write throughput was decreased most recently for this table. The time is expressed in the number of seconds that have elapsed since 00:00:00, January 1, 1970 (UTC).

- **table_options**: the TTL, MaxVersions, and MaxTimeDeviation configurations of the table.
 - **time_to_live**: the retention time of data. Unit: seconds.
 - The latest version of Table Store API supports automatic data expiration. If you do not want data to expire, set `time_to_live` to -1.
 - Table Store checks whether data has expired based on the data timestamp, current time, and table TTL. If the difference between the current time and the data timestamp is greater than the table TTL, data expires and is cleared by the Table Store server. For more information about the data timestamp, see [Preface](#).
 - After the TTL is set, the server checks whether data expires based on the data timestamp. If you specify a timestamp when writing data and the timestamp is significantly different from the actual time, unexpected data expiration may occur. For example, if the specified data timestamp is too small, the data may expire and be deleted as soon as it is written. If the specified timestamp is too large, the data may not expire when required. Therefore, when writing data with the TTL specified, make sure that you set a reasonable timestamp.
 - **max_versions**: the maximum number of versions retained by each attribute column.
 - The latest version of Table Store API supports data models of multiple versions. For more information, see [Preface](#). The `max_versions` parameter is used to specify the maximum number of data versions saved by each attribute column. If a new version causes the total number of versions to exceed the value of `max_versions`, the server deletes the earliest version to ensure that the number of versions do not exceed this value.
 - **deviation_cell_version_in_sec**: the maximum deviation allowed between the specified version and the current system time when data is written to the version. Unit: seconds.
 - Table Store supports multiple versions. By default, the system generates a new version number when writing new data. The version number is a timestamp corresponding to the write time in milliseconds. The TTL feature uses this timestamp to determine if data has expired. In addition, you can specify the data writing timestamp. If you set an extremely small timestamp, which deviates from the current time by more than the TTL set for the table,

the written data immediately expires. To prevent immediate data expiration, Table Store provides the `deviation_cell_version_in_sec` parameter to limit the permissible deviation between the data writing timestamp and the current system time. The value of this parameter is expressed in seconds. You can specify this parameter when creating a table and modify it by using the `UpdateTable` operation.

- `stream_details`: the Stream information of the table.
 - `enable_stream`: indicates whether Stream is enabled for the table.
 - `stream_id`: the Stream ID of the table.
 - `expiration_time`: the expiration time of the Stream data of the table. Unit: hours. Earlier modification records are deleted.
 - `last_enable_time`: the time when Stream was enabled most recently.

Example

The following code provides an example of how to obtain the description of a table:

```
$ result = $ client -> describeTable ([
    ' table_name ' => ' mySampleTable ',
]);
var_dump ($ result );
```

8.4.7 Create a search index

CreateSearchIndex

If a table needs to support complex queries, you must create a search index for the table. When creating a search index, you must specify the `table_name`, `index_name`, and `index_meta` parameters.

- `table_name`: the name of the table for which the search index is created.
- `index_name`: the name of the search index to be created.

- **index_meta**: the index schema, including the **index_setting**, **field_schemas**, and **index_sort** parameters.
- **field_schemas**: the list of **FieldSchemas**. Each **FieldSchema** contains the following parameters:
 - (Required) **field_name**: String. The name of the field to be indexed, that is, the column name. The field can be a primary key column or an attribute column.
 - (Required) **field_type**: **FieldTypeConst::XXX**. For more information, see the description of field types for a search index.
 - (Optional) **is_array**: Boolean. Specifies whether the column is an array. A value of true indicates that the column is an array. Data written to the column must be a JSON array, such as ["a","b","c"]. You do not need to explicitly specify this parameter for nested columns because they are arrays.
 - (Optional) **index**: Boolean. Specifies whether to index the column.
 - (Optional) **index_options**: the index configuration options.
 - (Optional) **analyzer**: the tokenizer.
 - (Optional) **enable_sort_and_agg**: Boolean. Specifies whether to enable sorting and statistics collection.
 - (Optional) **store**: Boolean. Specifies whether to store the values of the field in the search index. A value of true indicates that you can read the values of the field directly from the search index without the need to query the table. This optimizes query performance.
- (Optional) **index_setting**:
 - (Optional, advanced feature) **routing_fields**: the custom routing fields. You can specify some primary key columns as routing fields. Table Store distributes data that is written to a search index to different partitions based

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

- (Optional) `index_sort`: specifies how the index is sorted. The index is sorted in the same way as the primary key of the table by default.

■ `sorters`: The array of sorting methods. Valid values: `primary_key_sort` and `field_sort`.

■ `primary_key_sort`:

■ `order`: Default value: `SortOrderConst::SORT_ORDER_ASC`.

■ `field_sort`: specifies a field for index sorting.

■ `field_name`: the field for index sorting.

■ `order`: Default value: `SortOrderConst::SORT_ORDER_ASC`.

Method

```
/**
 * Create a search index .
 * @ api
 *
 * @ param [] $ request
 *      The request parameters , including the
table name and index configurat ion .
 * @ return [] The response .
 * @ throws OTSClientE xception The exception that is
thrown when a parameter check error occurs or the
Table Store server returns a verificati on error .
 * @ throws OTSServerE xception The exception that is
thrown when the Table Store server returns an error .
 * @ example " src / examples / CreateSear chIndex . php "
 */
public function createSear chIndex ( array $ request )
```

Example

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' schema ' => array (
        ' field_sche mas ' => array (
            array (
                ' field_name ' => ' keyword ',
                ' field_type ' => FieldTypeC onst :: KEYWORD ,
                ' index ' => true ,
                ' enable_sor t_and_agg ' => true ,
                ' store ' => true ,
                ' is_array ' => false
            ),
            array (
                ' field_name ' => ' text ',
                ' field_type ' => FieldTypeC onst :: TEXT ,
                ' analyzer ' => ' single_wor d ',
                ' index ' => true ,
```

```

        'enable_sort_and_agg' => false ,
        'store' => true ,
        'is_array' => false
    ),
    array (
        'field_name' => 'geo ',
        'field_type' => FieldTypeC::GEO_POINT ,
        'index' => true ,
        'index_options' => 'DOCS ',
        'enable_sort_and_agg' => true ,
        'store' => true ,
        'is_array' => false
    ),
    array (
        'field_name' => 'long ',
        'field_type' => FieldTypeC::LONG ,
        'index' => true ,
        'enable_sort_and_agg' => true ,
        'store' => true ,
        'is_array' => false
    ),
    array (
        'field_name' => 'double ',
        'field_type' => FieldTypeC::DOUBLE ,
        'index' => true ,
        'enable_sort_and_agg' => true ,
        'store' => true ,
        'is_array' => false
    ),
    array (
        'field_name' => 'boolean ',
        'field_type' => FieldTypeC::BOOLEAN ,
        'index' => true ,
        'enable_sort_and_agg' => false ,
        'store' => true ,
        'is_array' => false
    ),
    array (
        'field_name' => 'array ',
        'field_type' => FieldTypeC::KEYWORD ,
        'index' => true ,
        'enable_sort_and_agg' => false ,
        'store' => true ,
        'is_array' => true
    ),
    array (
        'field_name' => 'nested ',
        'field_type' => FieldTypeC::NESTED ,
        'index' => false ,
        'enable_sort_and_agg' => false ,
        'store' => false ,
        'field_schemas' => array (
            array (
                'field_name' => 'nested_key word ',
                'field_type' => FieldTypeC::KEYWORD
            ),
            array (
                'index' => false ,
                'enable_sort_and_agg' => false ,
                'store' => false ,
                'is_array' => false
            )
        )
    ),
),
),
),

```

```

        'index_setting' => array (
            'routing_fields' => array ("pk1 ")
        ), "index_sort" => array (// You cannot specify this
// parameter for nested columns .
//         array (
//             'field_sort' => array (
//                 'field_name' => 'keyword ',
//                 'order' => SortOrderConstant::SORT_ORDER
// _ASC ,
//                 'mode' => SortModeConstant::SORT_MODE_AVG
//             )
//         ),
//         array (
//             'pk_sort' => array (
//                 'order' => SortOrderConstant::SORT_ORDER
// _ASC
//             )
//         ),
//     )
// );
$response = $otsClient->createSearchIndex();

```

8.4.8 Obtain search indexes of a table

ListSearchIndex

You can call the ListSearchIndex operation to obtain all search indexes created for a table.

Method

```

* Obtain all search indexes created for a table .
* @api
*
* @param [] $request
*     The request parameter , which is the
table name .
* @return [] The response .
* @throws OTSClientException The exception that is
thrown when a parameter check error occurs or the
Table Store server returns a verification error .
* @throws OTSServerException The exception that is
thrown when the Table Store server returns an error .
* @example "src / examples / ListSearchIndex . php "
*/
public function listSearchIndex ( array $request )

```

Example

```

$request = array (
    'table_name' => 'php_sdk_test ',
);

```

```
$ response = $ otsClient -> listSearch Index ($ request );
```

8.4.9 Query the description of a search index

DescribeSearchIndex

You can call the DescribeSearchIndex operation to query the description of a search index for a table, including fields and configurations of the search index.

Method

```
/**
 * Obtain the description of a search index in
 * a table.
 * @api
 *
 * @param [] $ request
 * The request parameter, which is the
 * table name.
 * @return [] The response.
 * @throws OTSClientException The exception that is
 * thrown when a parameter check error occurs or the
 * Table Store server returns a verification error.
 * @throws OTSServerException The exception that is
 * thrown when the Table Store server returns an error.
 * @example "src / examples / DescribeSearchIndex . php "
 * 20
 */
public function describeSearchIndex ( array $ request )
```

Example

```
$ request = array (
    ' table_name ' => ' php_sdk_test ',
    ' index_name ' => ' php_sdk_test_search_index '
);
$ response = $ otsClient -> describeSearchIndex ($ request );
```

8.4.10 Delete a search index

DeleteSearchIndex

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

Method

```

 * Delete a search index.
 * @api
 *
 * @param [] $ request
 * The request parameter, which is the
 * table name.
 * @return [] The response.
```

```

    * @throws OTSClientException The exception that is
    thrown when a parameter check error occurs or the
    Table Store server returns a verification error .
    * @throws OTSServerException The exception that is
    thrown when the Table Store server returns an error .
    * @example " src / examples / DeleteSearchIndex . php "
    */
    public function deleteSearchIndex ( array $ request )

```

Example

```

$ request = array (
    ' table_name ' => ' php_sdk_test ',
    ' index_name ' => ' php_sdk_test_search_index '
);
$ response = $ otsClient -> deleteSearchIndex ( $ request );

```

8.4.11 Create a global secondary index

CreateIndex

You can call the CreateIndex operation to create a global secondary index for an existing table.

Method

```

/**
 * Create a global secondary index .
 * @api
 *
 * @param [] $ request
 * The request parameter , which is the
table name .
 * @return [] The response .
 * @throws OTSClientException The exception that is
thrown when a parameter check error occurs or the
Table Store server returns a verification error .
 * @throws OTSServerException The exception that is
thrown when the Table Store server returns an error .
 * @example " src / examples / CreateIndex . php " 20
 */
public function createIndex ( array $ request )

```

Example

```

$ request = array (
    ' table_name ' => ' MainTableName ',
    ' index_meta ' => array (
        ' name ' => self :: $ indexName1 ,
        ' primary_key ' => array ( ' col1 ' ),
        ' defined_column ' => array ( ' col2 ' )
    )
);

```

```
$ this -> otsClient -> createIndex ($ request );
```

CreateTable

You can call the CreateTable operation to create a global secondary index when creating a table.

Example

```
$ request = array (
    'table_meta' => array (
        'table_name' => self::$tableName, // The table name
        'primary_key_schema' => array (
            array ('PK0', PrimaryKey TypeConst::CONST_INTEGER),
            array ('PK1', PrimaryKey TypeConst::CONST_STRING)
        ),
        'defined_column' => array (
            array ('col1', DefinedColumnTypeConst::DCT_STRING),
            array ('col2', DefinedColumnTypeConst::DCT_INTEGER)
        ),
        'reserved_throughput' => array (
            'capacity_unit' => array (
                'read' => 0, // Set both the reserved read
                'write' => 0 // throughput and reserved write throughput to 0.
            )
        ),
        'table_options' => array (
            'time_to_live' => -1, // The data lifecycle.
            // Unit: seconds. A value of -1 indicates that data
            // never expires.
            'max_versions' => 1, // The maximum number of
            // data versions.
            'deviation_cell_version_in_sec' => 86400 // The
            // valid data version offset. Unit: seconds.
        ),
        'index_metadata' => array (
            array (
                'name' => self::$indexName1,
                'primary_key' => array ('col1'),
                'defined_column' => array ('col2')
            ),
            array (
                'name' => self::$indexName2,
                'primary_key' => array ('PK1'),
                'defined_column' => array ('col1', 'col2')
            )
        )
    );
```

```
$ this -> otsClient -> createTable ( $ request );
```

8.4.12 Delete a global secondary index

DropIndex

You can call the DropIndex operation to delete a global secondary index.

Method

```
/**
 * Delete a global secondary index .
 * @ api
 *
 * @ param [] $ request
 *         The request parameter , which is the
 *         table name .
 * @ return [] The response .
 * @ throws OTSClientException The exception that is
 *         thrown when a parameter check error occurs or the
 *         Table Store server returns a verification error .
 * @ throws OTSServerException The exception that is
 *         thrown when the Table Store server returns an error .
 * @ example " src / examples / DropIndex . php "
 */
public function dropIndex ( array $ request )
```

Example

```
$ request = array (
    ' table_name ' => ' MainTableName ',
    ' index_name ' => ' IndexName '
);
$ otsClient -> dropIndex ( $ request );
```

8.5 Table operations

The Table Store SDK provides CreateTable, ListTable, DeleteTable, UpdateTable, DescribeTable, and ComputeSplitsBySize for table operations.

CreateTable

API: [CreateTable](#)

You can call this operation to create a table. When creating a table in Table Store, you must specify TableMeta and TableOptions. You can also set ReservedThroughput.

After you create the table, the server loads splits of the table to a specified node.

Therefore, you must wait several seconds before reading from or writing to the table.

Otherwise, the system throws an exception. Note: If the system throws an exception, the SDK automatically retries the read or write operation.

Operation

```

/**
 * Create a table . Set the number , name , sequence
 , and type of primary key columns , reserved read /
 write throughput , time to live ( TTL ), and Stream
 options .
 * @ api
 * @ param [] $ request Request parameters .
 * @ return [] The response is empty . If the
 CreateTable e operation succeeds , no message is returned
 . An empty array is returned here to be consistent
 with other operations .
 * @ throws OTSClientException Indicates that a
 parameter check error occurs or the server returns a
 verification error .
 * @ throws OTSServerException Indicates that the
 Table Store server returns an error .
 */
public function createTable ( array $ request );

```

Request format

```

$ result = $ client -> createTable ( [
    ' table_meta ' => [ // REQUIRED
        ' table_name ' => '< string >',
        ' primary_key_schema ' => [
            ['< string >', < PrimaryKey Type >],
            ['< string >', < PrimaryKey Type >],
            ['< string >', < PrimaryKey Type >, < PrimaryKey Option
        >]
    ],
    ],
    ' reserved_throughput ' => [ // REQUIRED
        ' capacity_unit ' => [
            ' read ' => < integer >,
            ' write ' => < integer >
        ]
    ],
    ' table_options ' => [ // REQUIRED
        ' time_to_live ' => < integer >,
        ' max_versions ' => < integer >,
        ' deviation_cell_version_in_sec ' => < integer >
    ],
    ' stream_spec ' => [
        ' enable_stream ' => true || false ,
        ' expiration_time ' => < integer >
    ]
]);

```

Request format description

- **table_meta**: required. This parameter specifies the table name and schema of the primary key.
 - **table_name**: the name of the table.
 - **primary_key_schema**: the schema of the primary key of the table.
- The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
- The first primary key column serves as the partition key. Data with the same partition key value is in the same partition. Therefore, we recommend that you have no more than 10 GB of data with the same partition key value. Otherwise, individual partitions are too large and cannot be split. We also recommend that you distribute read and write operations evenly among different partitions to facilitate load balancing.
- When creating a table, you only need to define the primary key of the table. Attribute columns do not need to be defined. Different rows can have different data columns and the names of attribute columns can be specified when data is written. Therefore, Table Store is suitable for the storage of semi-structured data. You can dynamically add new data columns during business development.
- Each item consists of the primary key name, primary key type (PrimaryKeyType), and primary key settings (PrimaryKeyOption, which is optional).
- PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), and BINARY types.
- PrimaryKeyOption can be set to PrimaryKeyOptionConst::CONST_PK_AUTO_INCR, which indicates the auto-increment column. For more information, see Auto-increment primary key column.
- **reserved_throughput**: required. This parameter specifies the reserved read/write throughput, which is related to billing.
 - **capacity_unit**: When the reserved read/write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds

the reserved volume is billed on a Pay-As-You-Go basis. By default, the reserved read/write throughput is 0 and all traffic is billed on a Pay-As-You-Go basis. If you want to set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can set the reserved read/write throughput of capacity instances only to 0. These instances do not allow you to reserve read/write throughput.

- **read**: the reserved read throughput.

- **write**: the reserved write throughput.

- **table_options**: required. This parameter contains the TTL, MaxVersions, and MaxTimeDeviation configurations of the table.

- **time_to_live**: the retention time of data, in seconds.

- The latest API version of Table Store supports automatic data expiration. If you do not want data to expire, set `time_to_live` to -1.

- Table Store checks whether data has expired based on the data timestamp, current time, and table TTL. If the difference between the current time and the data timestamp is greater than the table TTL, data expires and is cleared by the Table Store server. For more information about the data timestamp, see [Preface](#).

- After the TTL is set, the server checks whether data expires based on the data timestamp. If you specify a timestamp when writing data and the timestamp is significantly different from the actual time, unexpected data expiration may occur. For example, if the specified data timestamp is too small, the data may expire and be deleted as soon as it is written. If the specified timestamp is too large, the data may not expire when required. Therefore, when writing data with the TTL specified, make sure that you set a reasonable timestamp.

- **max_versions**: the maximum number of versions retained by each attribute column.

- The latest API version of Table Store supports data models of multiple versions. For more information, see [Preface](#). The `max_versions` parameter is used to specify the maximum number of data versions saved by each attribute column. If a new version causes the total number of versions to exceed the

value of `max_versions`, the server deletes the earliest version to ensure that the number of versions do not exceed this value.

- `deviation_cell_version_in_sec`: the maximum deviation allowed between the specified version and the current system time when data is written to the version, in seconds.

■ Table Store supports multiple versions. By default, the system generates a new version number when writing new data. The version number is a timestamp corresponding to the write time in milliseconds. The TTL feature uses this timestamp to determine if data has expired. In addition, you can specify the data writing timestamp. If you set an extremely small timestamp, which deviates from the current time by more than the TTL set for the table, the written data immediately expires. To prevent immediate data expiration, Table Store provides the `deviation_cell_version_in_sec` parameter to limit the permissible deviation between the data writing timestamp and the current system time. The value of this parameter is in seconds. You can specify this parameter when creating a table and modify it by using the `UpdateTable` operation.

- `stream_spec`: optional. This parameter specifies the Stream-related settings.
 - `enable_stream`: indicates whether to enable Stream.
 - `expiration_time`: the expiration time of the Stream data of the table, in hours. Earlier modification records are deleted.

Result format

```
[]
```

Result format description

The response is empty. If an error occurs, the system throws an exception.

Example

The following code provides an example of how to create a table with three primary key columns. Set the reserved read/write throughput to 0 and TTL to -1, and enable Stream.

```
// Create a schema for the primary key columns
, including the number , names , and types of the
primary key columns
```

```

// The first primary key column (partition column
) is named PK0 and belongs to the integer type .
// The second primary key column is named PK1
and belongs to the string type .
// The third primary key column is named PK2
and belongs to the binary type .
$ result = $ client -> createTable ( [
    ' table_meta ' => [
        ' table_name ' => ' SampleTable ',
        ' primary_key_schema ' => [
            [ ' PK0 ', PrimaryKey TypeConst :: CONST_INTEGER ],
            [ ' PK1 ', PrimaryKey TypeConst :: CONST_STRING ],
            [ ' PK2 ', PrimaryKey TypeConst :: CONST_BINARY ]
        ]
    ],
    ' reserved_throughput ' => [
        ' capacity_unit ' => [
            ' read ' => 0 ,
            ' write ' => 0
        ]
    ],
    ' table_options ' => [
        ' time_to_live ' => - 1 ,
        ' max_versions ' => 2 ,
        ' deviation_cell_version_in_sec ' => 86400
    ],
    ' stream_spec ' => [
        ' enable_stream ' => true ,
        ' expiration_time ' => 24
    ]
]);

```

ListTable

API: [ListTable](#)

You can call this operation to obtain the names of all tables created in the current instance.

Operation

```

/**
 * Obtain the names of all tables created in the
current instance .
 * @api
 * @param [] $ request Request parameters , which are
empty .
 * @return [] Response .
 * @throws OTSClientException Indicates that a
parameter check error occurs or the server returns a
verification error .
 * @throws OTSServerException Indicates that the
Table Store server returns an error .
 */
public function listTable ( array $ request );

```

Request format

```
$ result = $ client -> listTable ([]);
```

Request format description

Currently, no parameter is required in the request.

Result format

```
[  
    '< string >',  
    '< string >',  
    '< string >'  
]
```

Result format description

The result is a list of strings. Each item indicates a table name.

Example

The following code provides an example of how to obtain the names of all tables in an instance:

```
$ result = $ otsClient -> listTable ([]);
```

UpdateTable

API: [UpdateTable](#)

You can call this operation to update ReservedThroughput, TableOptions, and StreamSpecification of a table.

For more information about ReservedThroughput, TableOptions, and StreamSpecification, see the description of the CreateTable operation. Currently, the interval between attempts to modify the value of ReservedThroughput is a minimum of 1 minute.

Operation

```
/**  
 * Update a table, including the reserved read /  
 write throughput, configuration information, and Stream  
 configuration of the table.  
 * This operation can be used to increase or  
 decrease the reserved read / write throughput.  
 * @api
```

```

    * @param [] $ request Request parameters .
    * @return [] Response .
    * @throws OTSClientException Indicates that a
parameter check error occurs or the server returns a
verificati on error .
    * @throws OTSServerErrorException Indicates that the
Table Store server returns an error .
    */
    public function updateTable ( array $ request );

```

Request format

```

$result = $ client -> updateTable ( [
    ' table_name ' => '< string >',           // REQUIRED
    ' reserved_throughput ' => [
        ' capacity_unit ' => [
            ' read ' => < integer >,
            ' write ' => < integer >
        ]
    ],
    ' table_options ' => [
        ' time_to_live ' => < integer >,
        ' max_versions ' => < integer >,
        ' deviation_cell_version_in_sec ' => < integer >
    ],
    ' stream_spec ' => [
        ' enable_stream ' => true || false ,
        ' expiration_time ' => < integer >
    ]
]);

```

Request format description

- The difference between UpdateTable and CreateTable is the TableMeta parameter . Except TableMeta, all other parameters are defined and set in the same way as those of CreateTable. Except table_name, all other parameters are optional.
- table_name: required. This parameter specifies the name of the table.
- reserved_throughput: optional. This parameter specifies the reserved read/write throughput, which is related to billing.
 - capacity_unit: When the reserved read/write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds the reserved volume is billed on a Pay-As-You-Go basis. By default, the reserved read/write throughput is 0 and all traffic is billed on a Pay-As-You-Go basis. If you want to set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can

set the reserved read/write throughput of capacity instances only to 0. These instances do not allow you to reserve read/write throughput.

- read: the reserved read throughput.

- write: the reserved write throughput.

- **table_options**: optional. This parameter contains the TTL, MaxVersions, and MaxTimeDeviation configurations of the table.

- **time_to_live**: the retention time of data, in seconds.

- The latest API version of Table Store supports automatic data expiration. If you do not want data to expire, set **time_to_live** to -1.

- Table Store checks whether data has expired based on the data timestamp, current time, and table TTL. If the difference between the current time and the data timestamp is greater than the table TTL, data expires and is cleared by the Table Store server. For more information about the data timestamp, see [Preface](#).

- After the TTL is set, the server checks whether data expires based on the data timestamp. If you specify a timestamp when writing data and the timestamp is significantly different from the actual time, unexpected data expiration may occur. For example, if the specified data timestamp is too small, the data may expire and be deleted as soon as it is written. If the specified timestamp is too large, the data may not expire when required. Therefore, when writing data with the TTL specified, make sure that you set a reasonable timestamp.

- **max_versions**: the maximum number of versions retained by each attribute column.

- The latest API version of Table Store supports data models of multiple versions. For more information, see [Preface](#). The **max_versions** parameter is used to specify the maximum number of data versions saved by each attribute column. If a new version causes the total number of versions to exceed the

value of `max_versions`, the server deletes the earliest version to ensure that the number of versions do not exceed this value.

- `deviation_cell_version_in_sec`: the maximum deviation allowed between the specified version and the current system time when data is written to the version, in seconds.

■ Table Store supports multiple versions. By default, the system generates a new version number when writing new data. The version number is a timestamp corresponding to the write time in milliseconds. The TTL feature uses this timestamp to determine if data has expired. In addition, you can specify the data writing timestamp. If you set an extremely small timestamp, which deviates from the current time by more than the TTL set for the table, the written data immediately expires. To prevent immediate data expiration, Table Store provides the `deviation_cell_version_in_sec` parameter to limit the permissible deviation between the data writing timestamp and the current system time. The value of this parameter is in seconds. You can specify this parameter when creating a table and modify it by using the `UpdateTable` operation.

- `stream_spec`: optional. This parameter specifies the Stream-related settings.
 - `enable_stream`: indicates whether to enable Stream.
 - `expiration_time`: the expiration time of the Stream data of the table, in hours. Earlier modification records are deleted.

Result format

```
[
  ' capacity_u nit_detail s ' => [
    ' capacity_u nit ' => [
      ' read ' => < integer >,
      ' write ' => < integer >
    ],
    ' last_incre ase_time ' => < integer >,
    ' last_decre ase_time ' => < integer >
  ],
  ' table_opti ons ' => [
    ' time_to_li ve ' => < integer >,
    ' max_versio ns ' => < integer >,
    ' deviation_ cell_versi on_in_sec ' => < integer >
  ],
  ' stream_det ails ' => [
    ' enable_str eam ' => true || false ,
    ' stream_id ' => '< string >',
    ' expiration _time ' => < integer >,
    ' last_enabl e_time ' => < integer >
  ]
]
```

]

Result format description

- **capacity_unit_details**: the reserved read/write throughput configuration, which is related to billing.
 - **capacity_unit**: When the reserved read/write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds the reserved volume is billed on a Pay-As-You-Go basis. By default, the reserved read/write throughput is 0 and all traffic is billed on a Pay-As-You-Go basis. If you want to set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can set the reserved read/write throughput of capacity instances only to 0. These instances do not allow you to reserve read/write throughput.
 - **read**: the reserved read throughput.
 - **write**: the reserved write throughput.
 - **last_increase_time**: the last time the reserved read/write throughput configuration was increased for this table, expressed in UTC time to the second.
 - **last_decrease_time**: the last time the reserved read/write throughput configuration was decreased for this table, expressed in UTC time to the second.
- **table_options**: the TTL, MaxVersions, and MaxTimeDeviation configurations of the table, which are the same as those in the request.
- **stream_details**: the Stream information of the table.
 - **enable_stream**: indicates whether Stream is enabled for the table.
 - **stream_id**: the Stream ID of the table.
 - **expiration_time**: the expiration time of the Stream data of the table, in hours. Earlier modification records are deleted.
 - **last_enable_time**: the last time that Stream was enabled.

Examples

The following code provides an example of how to update the read CU and write CU of a table to 1 and 2, respectively:

```
$ result = $ client -> updateTable ([
    'table_name' => 'SampleTable',
    'reserved_throughput' => [
        'capacity_unit' => [
```

```

        ' read ' => 1 , // The number of
read or write CUs consumed can be separately updated
        ' write ' => 2
    ]
});

```

The following code provides an example of how to set the TTL of a table to one day (86400s), retain two versions, and set the maximum deviation to 10s:

```

$ result = $ client -> updateTable ([
    ' table_name ' => ' SampleTable ',
    ' table_options ' => [
        ' time_to_live ' => 86400 ,
        ' max_versions ' => 2 ,
        ' deviation_cell_version_in_sec ' => 10
    ]
]);

```

The following code provides an example of how to enable Stream for the table and set the expiration time to 24 hours:

```

$ result = $ client -> updateTable ([
    ' table_name ' => ' SampleTable ',
    ' stream_spec ' => [
        ' enable_stream ' => true ,
        ' expiration_time ' => 24
    ]
]);

```

DescribeTable

API: [DescribeTable](#)

You can call this operation to query TableMeta, TableOptions, ReservedThroughputDetails, and StreamDetails of a table. TableMeta and TableOptions have already been described in the CreateTable section. Besides containing the reserved read/write throughput, ReservedThroughputDetails shows the last time the throughput was increased or decreased. StreamDetails shows the detailed information about Stream.

Operation

```

/**
 * Obtain information about a table, including
the primary key design and the reserved read / write
throughput .
 * @api
 * @param [] $ request Request parameters .
 * @return [] Response .

```

```

    * @throws OTSCClientException Indicates that a
    parameter check error occurs or the server returns a
    verification error .
    * @throws OTSServerException Indicates that the
    Table Store server returns an error .
    */
    public function describeTable ( array $ request );

```

Request format

```

$ result = $ client -> describeTable ([
    ' table_name ' => '< string >', // REQUIRED
]);

```

Request format description

- **table_name:** the name of the table.

Result format

```

[
    ' table_meta ' => [
        ' table_name ' => '< string >',
        ' primary_key_schema ' => [
            '< string >', < PrimaryKey Type >],
            '< string >', < PrimaryKey Type >, < PrimaryKey Option
        >]
    ],
    ' capacity_unit_detail ' => [
        ' capacity_unit ' => [
            ' read ' => < integer >,
            ' write ' => < integer >
        ],
        ' last_increase_time ' => < integer >,
        ' last_decrease_time ' => < integer >
    ],
    ' table_options ' => [
        ' time_to_live ' => < integer >,
        ' max_versions ' => < integer >,
        ' deviation_cell_version_in_sec ' => < integer >
    ],
    ' stream_details ' => [
        ' enable_stream ' => true || false ,
        ' stream_id ' => '< string >',
        ' expiration_time ' => < integer >,
        ' last_enable_time ' => < integer >
    ]
]

```

Result format description

- **table_meta:** the table name and schema of the primary key, which are the same as those specified when the table is created.

- **capacity_unit_details**: the reserved read/write throughput configuration, which is related to billing.
 - **capacity_unit**: When the reserved read/write throughput is greater than 0, you are charged based on the reserved volume and duration. Traffic that exceeds the reserved volume is billed on a Pay-As-You-Go basis. By default, the reserved read/write throughput is 0 and all traffic is billed on a Pay-As-You-Go basis. If you want to set a value greater than 0, we recommend that you read the documentation about Table Store billing to avoid unexpected fees. You can set the reserved read/write throughput of capacity instances only to 0. These instances do not allow you to reserve read/write throughput.
 - **read**: the reserved read throughput.
 - **write**: the reserved write throughput.
 - **last_increase_time**: the last time the reserved read/write throughput configuration was increased for this table, expressed in UTC time to the second.
 - **last_decrease_time**: the last time the reserved read/write throughput configuration was decreased for this table, expressed in UTC time to the second.
- **table_options**: the TTL, MaxVersions, and MaxTimeDeviation configurations of the table.
 - **time_to_live**: the retention time of data, in seconds.
 - The latest API version of Table Store supports automatic data expiration. If you do not want data to expire, set **time_to_live** to -1.
 - Table Store checks whether data has expired based on the data timestamp, current time, and table TTL. If the difference between the current time and the data timestamp is greater than the table TTL, data expires and is cleared by the Table Store server. For more information about the data timestamp, see [Preface](#).
 - After the TTL is set, the server checks whether data expires based on the data timestamp. If you specify a timestamp when writing data and the timestamp is significantly different from the actual time, unexpected data expiration may occur. For example, if the specified data timestamp is too small, the data may expire and be deleted as soon as it is written. If the specified timestamp

is too large, the data may not expire when required. Therefore, when writing data with the TTL specified, make sure that you set a reasonable timestamp.

- **max_versions**: the maximum number of versions retained by each attribute column.

■ The latest API version of Table Store supports data models of multiple versions. For more information, see [Preface](#). The **max_versions** parameter is used to specify the maximum number of data versions saved by each attribute column. If a new version causes the total number of versions to exceed the value of **max_versions**, the server deletes the earliest version to ensure that the number of versions do not exceed this value.

- **deviation_cell_version_in_sec**: the maximum deviation allowed between the specified version and the current system time when data is written to the version, in seconds.

■ Table Store supports multiple versions. By default, the system generates a new version number when writing new data. The version number is a timestamp corresponding to the write time in milliseconds. The TTL feature uses this timestamp to determine if data has expired. In addition, you can specify the data writing timestamp. If you set an extremely small timestamp, which deviates from the current time by more than the TTL set for the table, the written data immediately expires. To prevent immediate data expiration, Table Store provides the **deviation_cell_version_in_sec** parameter to limit the permissible deviation between the data writing timestamp and the current system time. The value of this parameter is in seconds. You can specify this parameter when creating a table and modify it by using the **UpdateTable** operation.

- **stream_details**: the Stream information of the table.
 - **enable_stream**: indicates whether Stream is enabled for the table.
 - **stream_id**: the Stream ID of the table.
 - **expiration_time**: the expiration time of the Stream data of the table, in hours. Earlier modification records are deleted.
 - **last_enable_time**: the last time that Stream was enabled.

Example

The following code provides an example of how to obtain the descriptive information of a table:

```
$ result = $ client -> describeTable ([
    ' table_name ' => ' mySampleTable ',
]);
var_dump ($ result );
```

DeleteTable

API: [DeleteTable](#)

You can call this operation to delete a specified table from an instance.

Operation

```
/**
 * Delete a table based on the table name .
 * @api
 * @param [] $ request Request parameters .
 * @return [] The response is empty . If the
DeleteTable operation succeeds , no message is returned
. An empty array is returned here to be consistent
with other operations .
 * @throws OTSClientException Indicates that a
parameter check error occurs or the server returns a
verification error .
 * @throws OTSServerErrorException Indicates that the
Table Store server returns an error .
 */
public function deleteTable ( array $ request );
```

Request format

```
$ result = $ client -> deleteTable ([
    ' table_name ' => '< string >', // REQUIRED
]);
```

Request format description

table_name: the name of the table.

Result format

```
[]
```

Result format description

The response is empty. If an error occurs, the system throws an exception.

Example

The following code provides an example of how to delete a table:

```
$ result = $ otsClient -> deleteTable ([
    ' table_name ' => ' MyTable '
]);
```

ComputeSplitsBySize

API: [ComputeSplitPointsBySize](#)

You can call this operation to logically split data in a table into several partitions whose sizes are close to the specified size, and return the split points between the partitions and prompt about hosts where the partitions reside. This operation is generally used for execution plans on compute engines, such as concurrency plans.

Operation

```
/**
 * Logically split data in a table into several
 * partitions whose sizes are close to the specified
 * size, and return the split points between the
 * partitions and prompt about hosts where the partitions
 * reside.
 * This operation is generally used for execution
 * plans on compute engines, such as concurrency plans.
 * @api
 * @param [] $ request Request parameters.
 * @return [] Response.
 * @throws OTSClientException Indicates that a
 * parameter check error occurs or the server returns a
 * verification error.
 * @throws OTSServerException Indicates that the
 * Table Store server returns an error.
 */
public function computeSplitPointsBySize ( array $
request )
```

Request format

```
$ result = $ client -> ComputeSplitPointsBySize ([
    ' table_name ' => '< string >', // REQUIRED
    ' split_size ' => < integer > // REQUIRED
]);
```

Request format description

- **table_name:** the name of the table.
- **split_size:** the approximate size of each split, in megabytes.

Result format

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => < integer >,
            'write' => < integer >
        ]
    ],
    'primary_key_schema' => [
        ['< string >', < PrimaryKey Type >],
        ['< string >', < PrimaryKey Type >, < PrimaryKey Option >]
    ],
    'splits' => [
        [
            'lower_bound' => [
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >],
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >]
            ],
            'upper_bound' => [
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >],
                ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >]
            ],
            'location' => '< string >'
        ],
        // ...
    ]
]
```

Result format description

- **consumed:** the number of CUs consumed by this operation.
 - **capacity_unit:** the number of read and write CUs consumed.
 - **read:** the number of read CUs consumed.
 - **write:** the number of write CUs consumed.
- **primary_key_schema:** the schema of the primary key of the table, which is the same as that specified when the table is created.
- **splits:** the split points between partitions.
 - **lower_bound:** the minimum value in the range of the primary key. Note: This parameter value can be used by GetRange.
 - Each item contains the primary name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType).
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, PrimaryKeyTypeConst::CONST_BINA

RY, `PrimaryKeyTypeConst::CONST_INF_MIN`, or `PrimaryKeyTypeConst::CONST_INF_MAX`, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, INF_MIN(-inf), and INF_MAX(inf) types.

- `upper_bound`: the maximum value in the range of the primary key. The format is the same as that of `lower_bound`.
- `location`: the prompt about the hosts where the split points reside. This parameter can be null.

Example

The following code provides an example of how to specify the size to split data:

```
$ result = $ client -> ComputeSplitsBySize ([
    ' table_name ' => ' MyTable ',
    ' split_size ' => 1
]);
foreach ($ result [' splits '] as $ split ) {
    print_r ($ split [' location ']);
    print_r ($ split [' lower_bound ']);
    print_r ($ split [' upper_bound ']);
}
```

8.6 Local transactions

StartLocalTransaction

You can call the `StartLocalTransaction` operation to create a local transaction (that is, a partition-level transaction) and obtain the transaction ID.

Method

```
/**
 * Create a transaction and obtain the transaction ID.
 * @api
 * @param [] $ request
 *      The request parameters, including the table name and partition key.
 * @return [] The response.
 * @throws OTSClientException The exception that is thrown when a parameter check error occurs or the Table Store server returns a verification error.
 * @throws OTSServerException The exception that is thrown when the Table Store server returns an error.
 * @example " src / examples / StartLocalTransaction.php
 * 50
 */
```

```
public function startLocal Transaction ( array $ request
)
```

Example

```
$ response = $ this -> otsClient -> startLocal Transaction (
    array (
        ' table_name ' => ' TransactionTable ',
        ' key ' => array ( // Primary keys : [ PK0 : INTEGER , PK1 :
        STRING ]
            array ( ' PK0 ', 0 )
        )
    )
);
```

After obtaining the transaction ID, you must pass it when calling the PutRow, UpdateRow, and BatchWriteRow operations.

```
$ updateRequest = array (
    ' table_name ' => ' TransactionTable ',
    ' condition ' => RowExistenceExpectationConst::CONST_IGNORED,
    ' primary_key ' => array (
        array ( ' PK0 ', 0 ),
        array ( ' PK1 ', ' 1 ' )
    ),
    ' update_of_attribute_columns ' => array (
        ' PUT ' => array (
            array ( ' attr0 ', ' new value ' )
        )
    ),
    ' transaction_id ' => $ response [ ' transaction_id ' ]
);
$ this -> otsClient -> updateRow ( $ updateRequest );
```

CommitTransaction

You can call the CommitTransaction operation to submit a transaction.

Method

```
/**
 * Submit a transaction .
 * @api
 *
 * @param [] $ request
 * The request parameter , which is the
table name .
 * @return [] The response .
 * @throws OTSClientException The exception that is
thrown when a parameter check error occurs or the
Table Store server returns a verification error .
 * @throws OTSServerException The exception that is
thrown when the Table Store server returns an error .
 * @example " src / examples / CommitTransaction . php " 50
 */
```

```
public function commitTransaction ( array $ request )
```

Example

```
$ this -> otsClient -> commitTransaction ( array (
    'transaction_id' => $ response ['transaction_id']
));
```

AbortTransaction

You can call the AbortTransaction operation to abort a transaction. All requests that contain the ID of this transaction are discarded.

Method

```
/**
 * Abort a transaction .
 * @api
 *
 * @param [] $ request
 * The request parameter , which is the
 * transaction ID .
 * @return [] The response .
 * @throws OTSClientException The exception that is
 * thrown when a parameter check error occurs or the
 * Table Store server returns a verification error .
 * @throws OTSServerException The exception that is
 * thrown when the Table Store server returns an error .
 * @example " src / examples / AbortTransaction . php " 20
 */
public function abortTransaction ( array $ request )
```

Example

```
$ this -> otsClient -> abortTransaction ( array (
    'transaction_id' => $ response ['transaction_id']
));
```

8.7 Single-row operations

The Table Store SDK provides the PutRow, GetRow, UpdateRow, and DeleteRow operations for single-row operations.

PutRow

PutRow

You can call this operation to insert a row of data. If the row does not exist, this operation inserts a row. If the row exists, this operation overwrites it. (This means that all columns and column values of all versions of the existing row are deleted).

When performing PutRow, you can use conditional update to set conditions for the existence of the row or conditions for values in certain columns of the row. For more information, see [Conditional update](#).

Operation

```
/**
 * Write a row of data. If the row already
 * exists, the data of the row is overwritten. The
 * number of the consumed capacity units (CUs) is
 * returned.
 * @api
 * @param [] $request Request parameters.
 * @return [] Response.
 * @throws OTSClientException Indicates that a
 * parameter check error occurs or the server returns a
 * verification error.
 * @throws OTSServerException Indicates that the
 * Table Store server returns an error.
 */
public function putRow ( array $request );
```

Request format

```
$result = $client->putRow ([
    'table_name' => '< string >', // REQUIRED
    'condition' => [
        'row_existence' => < RowExistence >,
        'column_condition' => < ColumnCondition >
    ],
    'primary_key' => [ // REQUIRED
        [< string >, < PrimaryKeyValue >],
        [< string >, < PrimaryKeyValue >],
        [< string >, < PrimaryKeyValue >, < PrimaryKeyType >]
    ],
    'attribute_columns' => [ // REQUIRED
        [< string >, < ColumnValue >],
        [< string >, < ColumnValue >, < ColumnType >],
        [< string >, < ColumnValue >, < ColumnType >, <
integer >]
    ],
    'return_content' => [
        'return_type' => < ReturnType >
    ]
]);
```

Request format description

- **table_name**: required. This parameter specifies the name of the table.
- **condition**: For more information, see [Conditional update](#).
 - **row_existence**: the row existence condition.
 - **column_condition**: the column-based condition.

- **primary_key:** required. This parameter specifies the primary key value of the row.
 - The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
 - Each item consists of the primary key name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType, which is optional).
 - The value of PrimaryKeyValue can be an integer or a string. This parameter can be set to null for an auto-increment primary key column.
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, PrimaryKeyTypeConst::CONST_BINARY, or PrimaryKeyTypeConst::CONST_PK_AUTO_INCR, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, and PK_AUTO_INCR types. PK_AUTO_INCR indicates the auto-increment column. For more information, see Auto-increment primary key column. If the type is INTEGER or STRING, it can be ignored. Otherwise, the type must be specified.
- **attribute_columns:** required. This parameter specifies the column attribute value.
 - Each item consists of the attribute name, attribute value (ColumnValue), column type (ColumnType, which is optional), and timestamp (which is optional).
 - ColumnType can be set to ColumnTypeConst::CONST_INTEGER, ColumnTypeConst::CONST_STRING, ColumnTypeConst::CONST_BINARY, ColumnTypeConst::CONST_BOOLEAN, or ColumnTypeConst::CONST_DOUBLE, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, BOOLEAN, and DOUBLE types. If the type is BINARY, it must be specified. Otherwise, the type can be ignored or set to null.
 - The timestamp is a 64-bit integer that represents the version of an attribute. This parameter is optional. If this parameter is not specified, the server time is used.
- **return_content:** the content to be returned.
 - **return_type:** Currently, only this parameter is required.
 - A value of ReturnTypeConst::CONST_PK indicates that the value of the auto-increment primary key column is returned.

Result format

[

```

    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKey Value>],
        ['<string>', <PrimaryKey Value>],
        ['<string>', <PrimaryKey Value>, <PrimaryKey Type>]
    ],
    'attribute_columns' => []
]

```

Result format description

- **consumed:** the number of CUs consumed by this operation.
 - **capacity_unit:** the number of read and write CUs consumed.
 - **read:** the number of read CUs consumed.
 - **write:** the number of write CUs consumed.
- **primary_key:** the primary key value, which is consistent with that in the request. If you set the return type to return the value of the auto-increment primary key, this primary key value is returned.
- **attribute_columns:** the attribute value, which is the same as that in the request. It is currently empty.

Example 1

The following code provides an example of how to write a row with 10 attribute columns and write one version of data for each column. The version number (timestamp) is specified by the server.

```

$ attr = array ();
for ($ i = 0 ; $ i < 10 ; $ i ++ ) {
    $ attr [] = [ ' Col '. $ i , $ i ];
}
$ request = [
    'table_name' => ' MyTable ',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE, // condition can be set to IGNORE, EXPECT_EXIST
    , or EXPECT_NOT_EXIST .
    'primary_key' => [ // The primary key .
        [ ' PK0 ', 123 ],
        [ ' PK1 ', ' abc ' ]
    ],
    'attribute_columns' => $ attr
];
$ response = $ otsClient -> putRow ( $ request );

```

Example 2

The following code provides an example of how to write a row with 10 attribute columns and write three versions of data for each column. The version number (timestamp) is specified by the client.

```
$attr = array();
$timestamp = getMicroTime();
for ($i = 0; $i < 10; $i++) {
    for ($j = 0; $j < 3; $j++) {
        $attr[] = ['Col '.$i, $j, null, $timestamp+$j];
    }
}
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE, // condition can be set to IGNORE, EXPECT_EXIST, or EXPECT_NOT_EXIST.
    'primary_key' => [ // The primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow($request);
```

Example 3

The following code provides an example of how to write data if the target row does not exist:

```
$attr = array();
$timestamp = getMicroTime();
for ($i = 0; $i < 10; $i++) {
    for ($j = 0; $j < 3; $j++) {
        $attr[] = ['Col '.$i, $j, null, $timestamp+$j];
    }
}
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST, // Configure the condition so that data is written if the target row does not exist.
    'primary_key' => [ // The primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow($request);
```

Example 4

The following code provides an example of how to write data if the target row already exists and the value of Col0 is greater than 100:

```
$ attr = array ();
$ timestamp = getMicroTime ();
for ($ i = 0 ; $ i < 10 ; $ i ++ ) {
    for ($ j = 0 ; $ j < 3 ; $ j ++ ) {
        $ attr [] = [ ' Col ' . $ i , $ j , null , $ timestamp + $ j ];
    }
}
$ request = [
    ' table_name ' => ' MyTable ',
    ' condition ' => [
        ' row_existence ' => RowExistenceExpectationConst::
CONST_EXPECT_EXIST, // Configure the condition so that
data is written if the target row exists .
        ' column_condition ' => [ // If the
condition is met , the data is updated .
            ' column_name ' => ' Col0 ',
            ' value ' => 100 ,
            ' comparator ' => ComparatorTypeConst:: CONST_GREATER_THAN
        ],
        ' primary_key ' => [ // The primary key .
            [ ' PK0 ', 123 ],
            [ ' PK1 ', ' abc ' ]
        ],
        ' attribute_columns ' => $ attr
    ];
$ response = $ otsClient -> putRow ( $ request );
```

GetRow

OperationsSummary

The GetRow operation specifies the table name and the primary key of a row. The two possible reading results are as follows:

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

Operation

```
/**
 * Read a row of data .
 * @ api
 * @ param [] $ request Request parameters .
 * @ return [] Response .
 * @ throws OTSClientException Indicates that a
parameter check error occurs or the server returns a
verification error .
 * @ throws OTSServerException Indicates that the
Table Store server returns an error .
 */
```

```
public function getRow ( array $ request );
```

Request format

```
$ result = $ client -> getRow ([
    ' table_name ' => '< string >', // REQUIRED
    ' primary_key ' => [ // REQUIRED
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >, < PrimaryKey Type >]
    ],
    ' max_versions ' => < integer >,
    ' time_range ' => [
        ' start_time ' => < integer >,
        ' end_time ' => < integer >,
        ' specific_time ' => < integer >
    ],
    ' start_column ' => '< string >',
    ' end_column ' => '< string >',
    ' token ' => '< string >',
    ' columns_to_get ' => [
        '< string >',
        '< string >',
        '//...
    ],
    ' column_filter ' => < ColumnCondition >
]);
```

Request format description

- **table_name:** required. This parameter specifies the name of the table.
- **primary_key:** required. This parameter specifies the primary key value of the row.
 - The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
 - Each item consists of the primary key name, primary key value (PrimaryKey Value), and primary key type (PrimaryKeyType, which is optional).
 - The value of PrimaryKeyValue can be an integer or a string.
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), and BINARY types. If the type is INTEGER or STRING, it can be ignored. Otherwise, the type must be specified.
- **max_versions:** the maximum number of versions from which data is read.

- **time_range**: the range of versions from which data is read. For more information, see [TimeRange](#).
 - **start_time**: the start timestamp, in milliseconds. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively. To query data within a range, specify **start_time** and **end_time**. The range is a left-closed right-open interval.
 - **end_time**: the end timestamp, in milliseconds. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively.
 - **specific_time**: the specific timestamp. To query data at a specific time, specify **specific_time**. You can set either **specific_time** or [**start_time**, **end_time**). The unit of this parameter is millisecond. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively.
- You must specify at least one of **max_versions** and **time_range**.
 - If you only specify **max_versions**, data of up to the specified number of versions is returned from the latest to the earliest.
 - If you only specify **time_range**, all data within the range is returned.
 - If you specify both **max_versions** and **time_range**, data of up to the specified number of versions within the version range is returned from the latest to the earliest.
- **columns_to_get**: the set of columns to be read. If you do not specify this parameter, all columns are read.
- **start_column**: the start column to be read, which is used for reading wide rows. The returned result includes the start column. The column names are sorted lexicographically. Assume that a table contains columns "a", "b", and "c". If the value of **start_column** is "b", the reading starts from column "b", and columns "b" and "c" are returned.
- **end_column**: the end column to be read, which is used for reading wide rows. The returned result does not include the end column. The column names are sorted lexicographically. Assume that a table contains columns "a", "b", and "c". If the value of **end_column** is "b", the reading ends at column "b", and column "a" is returned.
- **token**: the starting position of a wide row to be read next time. This parameter is currently unavailable.

- **column_filter**: the filtering condition. Only rows meeting the condition are returned. This parameter is similar to **column_condition** in **condition**. For more information, see [Filter](#).

Result format

```
[
  'consumed' => [
    'capacity_unit' => [
      'read' => <integer>,
      'write' => <integer>
    ]
  ],
  'primary_key' => [
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>, <PrimaryKey Type>]
  ],
  'attribute_columns' => [
    integer >] ['<string>', <ColumnValue>, <ColumnType>, <
    integer >] ['<string>', <ColumnValue>, <ColumnType>, <
    integer >] ['<string>', <ColumnValue>, <ColumnType>, <
  ],
  'next_token' => '<string>'
]
```

Result format description

- **consumed**: the number of CUs consumed by this operation.
 - **capacity_unit**: the number of read and write CUs consumed.
 - **read**: the number of read CUs consumed.
 - **write**: the number of write CUs consumed.
- **primary_key**: the primary key value, which is consistent with that in the request. If you set the return type to return the value of the auto-increment primary key, this primary key value is returned.
- **attribute_columns**: the attribute value.
 - Each item consists of the attribute name, attribute value (ColumnValue), attribute type (ColumnType), and timestamp.
 - ColumnType can be set to ColumnTypeConst::CONST_INTEGER, ColumnTypeConst::CONST_STRING, ColumnTypeConst::CONST_BINARY, ColumnTypeConst::CONST_BOOLEAN, or ColumnTypeConst::CONST_DOUBLE, which respectively

indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, BOOLEAN, and DOUBLE types.

- The timestamp is a 64-bit integer that represents the version of an attribute.
- The attributes in the returned result are sorted by attribute name lexicographically in ascending order. The versions of the attributes are sorted by timestamp in descending order.
- The order of the columns may be inconsistent with that of `columns_to_get` in the request.
- `next_token`: the starting position of a wide row to be read next time. This parameter is currently unavailable. (This parameter is currently unavailable.)
- If the row does not exist, the values of `primary_key` and `attribute_columns` are empty lists [].

Example 1

The following code provides an example of how to read the latest version of a column in a row:

```
$ request = [
    'table_name' => 'MyTable',
    'primary_key' => [ // The primary key .
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'max_versions' => 1, // Set this
    // parameter so that the latest version is read .
    'columns_to_get' => ['Col0'] // Set the
    // column to be read .
];
$response = $otsClient -> getRow ($ request );
```

Note:

- If you query a row of data, the system returns the data in all columns of the row. You can specify `columns_to_get` to enable the system to return the data in specific columns. For example, if you set `columns_to_get` to `Col0` and `Col1`, the system only returns the data in `Col0` and `Col1`.
- Conditional filtering is supported. For example, you can configure the system to return results only when the value of `Col0` is greater than 24.
- When both `columns_to_get` and `column_filter` are specified, the system first returns results based on the value of `columns_to_get`, and then filters the returned columns based on the value of `column_filter`.

- When a specified column does not exist, `pass_if_missing` determines the next action. The default value is `true`, indicating that the condition is valid if the column does not exist.

Example 2

The following code provides an example of how to set a filter:

```
$ request = [
    ' table_name ' => ' MyTable ',
    ' primary_key ' => [ // The primary key .
        [ ' PK0 ', 123 ],
        [ ' PK1 ', ' abc ' ]
    ],
    ' max_versions ' => 1 , // Set this
    // parameter so that the latest version is read .
    ' column_filter ' => [ // Set the
    // filter so that a row is returned if the value of
    // Col0 is 0 .
        ' column_name ' => ' Col0 ',
        ' value ' => 0 ,
        ' comparator ' => Comparator TypeConst :: CONST_EQUAL ,
        ' pass_if_missing ' => false // If a row
    // does not contain the Col0 column , the row is not
    // returned .
    ]
];
$response = $ otsClient -> getRow ( $ request );
```

UpdateRow

UpdateRow

You can call this operation to update a single row of data. If no row exists, a new row is added.

The update operations include writing a column, deleting a column, and deleting a version of a column.

An update operation may be performed in any of the following scenarios:

- Write a column value without specifying the version. Table Store automatically adds a version number to make sure that the version number is incremented.
- Write a column value with the specified version. If the column does not have the column value of this version, the data is inserted. Otherwise, the existing value is overwritten.
- Delete the column values of the specified version.
- Delete the column values of all versions for the entire column.

When performing `UpdateRow`, you can use conditional update to set conditions for the existence of the row or conditions for values in certain columns of the row. For more information, see [Conditional update](#).

Operation

```
/**
 * Update a row of data .
 * @ api
 * @ param [] $ request Request parameters .
 * @ return [] Response .
 * @ throws OTSClientException Indicates that a
parameter check error occurs or the server returns a
verificati on error .
 * @ throws OTSServerException Indicates that the
Table Store server returns an error .
 */
public function updateRow ( array $ request );
```

Request format

```
$ result = $ client -> updateRow ([
    ' table_name ' => '< string >', // REQUIRED
    ' condition ' => [
        ' row_existence ' => < RowExistence >,
        ' column_condition ' => < ColumnCondition >
    ],
    ' primary_key ' => [ // REQUIRED
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >, < PrimaryKey Type >]
    ],
    ' update_of_attribute_columns ' => [ // REQUIRED
        ' PUT ' => [
            ['< string >', < ColumnValue >],
            ['< string >', < ColumnValue >, < ColumnType >],
            ['< string >', < ColumnValue >, < ColumnType >, <
integer >]
        ],
        ' DELETE ' => [
            ['< string >', < integer >],
            ['< string >', < integer >],
            ['< string >', < integer >],
            ['< string >', < integer >]
        ],
        ' DELETE_ALL ' => [
            '< string >',
            '< string >',
            '< string >',
            '< string >'
        ],
    ],
    ' return_content ' => [
        ' return_type ' => < ReturnType >
    ]
]);
```

Request format description

- **table_name**: required. This parameter specifies the name of the table.
- **condition**: the condition for the operation to take effect. For more information, see [Conditional update](#).
 - **row_existence**: the row existence condition.
 - **column_condition**: the column-based condition.
- **primary_key**: required. This parameter specifies the primary key value of the row.
 - The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
 - Each item consists of the primary key name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType, which is optional).
 - The value of PrimaryKeyValue can be an integer or a string.
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), and BINARY types. If the type is INTEGER or STRING, it can be ignored. Otherwise, the type must be specified.
- **update_of_attribute_columns**: the modification on the row attributes. (This parameter is required. If none of the operations is performed, the corresponding item can be ignored.) The update operations are as follows:
 - **PUT**: indicates that if this column does not exist, a new column is added. If the column exists, it is overwritten. The format is the same as that of attribute_columns in PutRow.
 - Each item consists of the attribute name, attribute value (ColumnValue), column type (ColumnType, which is optional), and timestamp (which is optional).
 - ColumnType can be set to ColumnTypeConst::CONST_INTEGER, ColumnTypeConst::CONST_STRING, ColumnTypeConst::CONST_BINARY, ColumnTypeConst::CONST_BOOLEAN, or ColumnTypeConst::CONST_DOUBLE, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), BINARY

, BOOLEAN, and DOUBLE types. If the type is BINARY, it must be specified.

Otherwise, the type can be ignored or set to null.

- The timestamp is a 64-bit integer that represents the version of an attribute. This parameter is optional. If this parameter is not specified, the server time is used.

- DELETE: indicates that data of the specified version in the column is deleted. The timestamp must be specified.

- Each item consists of the attribute name and timestamp.

- The timestamp is a 64-bit integer that represents the attribute of the specified version.

- DELETE_ALL: indicates that data of all versions in the column is deleted. Note: Deleting all attribute columns of a row is not the same as deleting the row. To delete the row, use the DeleteRow operation.

- You only need to specify the attribute name.

- return_content: the content to be returned.

- return_type: Currently, only this parameter is required.

- A value of ReturnTypeConst::CONST_PK indicates that the value of the auto-increment primary key column is returned.

Result format

```
[
  'consumed' => [
    'capacity_unit' => [
      'read' => <integer>,
      'write' => <integer>
    ]
  ],
  'primary_key' => [
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>, <PrimaryKey Type>]
  ],
  'attribute_columns' => []
]
```

Result format description

- **consumed**: the number of CUs consumed by this operation.
 - **capacity_unit**: the number of read and write CUs consumed.
 - **read**: the number of read CUs consumed.
 - **write**: the number of write CUs consumed.
- **primary_key**: the primary key value, which is consistent with that in the request. If you set the return type to return the value of the auto-increment primary key, this primary key value is returned.
- **attribute_columns**: the attribute value, which is the same as that in the request. It is currently empty.

Example 1

The following code provides an example of how to update several columns, delete the specified version of a column, and delete the specified column:

```
$ request = [
    ' table_name ' => ' MyTable ',
    ' condition ' => RowExistenceExpectationConst::CONST_IGNORED,
    ' primary_key ' => [ // The primary key .
        [ ' PK0 ', 123 ],
        [ ' PK1 ', ' abc ' ]
    ],
    ' update_of_attribute_columns ' => [
        ' PUT ' => [ // Update several
columns .
            [ ' Col0 ', 100 ],
            [ ' Col1 ', ' Hello ' ],
            [ ' Col2 ', ' a binary ', ColumnTypeConst::CONST_BINARY ],
            [ ' Col3 ', 100, null, 1526418378, 526 ]
        ],
        ' DELETE ' => [ // Delete a version
of a column .
            [ ' Col10 ', 1526418378, 526 ]
        ],
        ' DELETE_ALL ' => [
            [ ' Col11 ' // Delete a column .
        ]
    ]
];
$response = $otsClient->updateRow($request);
```

Note:

- UpdateRow also supports conditional statements.

Example 2

The following code provides an example of how to set update conditions:

```
$ request = [
    'table_name' => ' MyTable ',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => [ // The primary key .
        [' PK0 ', 123 ],
        [' PK1 ', ' abc ' ]
    ],
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST, // Update the row if it exists .
        'column_filter' => [
            // Update the row when the value of Col0
            is greater than 100 .
            'column_name' => ' Col0 ',
            'value' => 100 ,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ]
    ],
    'update_of_attribute_columns' => [
        'PUT' => [ // Update several
            columns .
                [' Col0 ', 100 ],
                [' Col1 ', ' Hello ' ],
                [' Col2 ', ' a binary ', ColumnTypeConst::CONST_BINARY ],
                [' Col3 ', 100 , null , 1526418378 526 ]
            ],
        'DELETE' => [ // Delete a version
            of a column .
                [' Col10 ', 1526418378 526 ]
            ],
        'DELETE_ALL' => [
            ' Col11 ' // Delete a column .
        ]
    ]
];
```

DeleteRow

DeleteRow

You can call this operation to delete a row. No error is reported regardless of whether the specified row exists.

When performing DeleteRow, you can use conditional update to set conditions for the existence of the row or conditions for values in certain columns of the row. For more information, see [Conditional update](#).

Operation

```
/**
 * Delete a row of data .
 * @api
```

```

    * @param [] $request Request parameters .
    * @return [] Response .
    * @throws OTSClientException Indicates that a
parameter check error occurs or the server returns a
    verification error .
    * @throws OTSServerException Indicates that the
Table Store server returns an error .
    */
    public function deleteRow ( array $request );

```

Request format

```

$result = $client->deleteRow ([
    'table_name' => '< string >', // REQUIRED
    'condition' => [
        'row_existence' => < RowExistence >,
        'column_condition' => < ColumnCondition >
    ],
    'primary_key' => [ // REQUIRED
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >, < PrimaryKey Type >]
    ],
    'return_content' => [
        'return_type' => < ReturnType >
    ]
]);

```

Request format description

- **table_name:** required. This parameter specifies the name of the table.
- **condition:** the condition for the operation to take effect. For more information, see [Conditional update](#).
 - **row_existence:** the row existence condition.
 - **column_condition:** the column-based condition.
- **primary_key:** required. This parameter specifies the primary key value of the row.
 - The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
 - Each item consists of the primary key name, primary key value (PrimaryKey Value), and primary key type (PrimaryKeyType, which is optional).
 - The value of PrimaryKeyValue can be an integer or a string.
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which respectively indicate the INTEGER, STRING (UTF-8 encoded string

), and BINARY types. If the type is INTEGER or STRING, it can be ignored.

Otherwise, the type must be specified.

- **return_content**: the content to be returned.
- **return_type**: Currently, only this parameter is required.
 - A value of `ReturnTypeConst::CONST_PK` indicates that the value of the auto-increment primary key column is returned.

Result format

```
[
  'consumed' => [
    'capacity_unit' => [
      'read' => <integer>,
      'write' => <integer>
    ],
  ],
  'primary_key' => [
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>, <PrimaryKey Type>]
  ],
  'attribute_columns' => []
]
```

Result format description

- **consumed**: the number of CUs consumed by this operation.
 - **capacity_unit**: the number of read and write CUs consumed.
 - **read**: the number of read CUs consumed.
 - **write**: the number of write CUs consumed.
- **primary_key**: the primary key value, which is consistent with that in the request. If you set the return type to return the value of the auto-increment primary key, this primary key value is returned.
- **attribute_columns**: the attribute value, which is the same as that in the request. It is currently empty.

Example 1

The following code provides an example of how to delete a row of data:

```
$request = [
  'table_name' => 'MyTable',
  'condition' => RowExistenceExpectationConst::CONST_IGNORED,
  'primary_key' => [ // The primary key.
    ['PK0', 123],
```

```

        [' PK1 ', ' abc ' ]
    ],
    'return_content' => [
        'return_type' => ReturnType Const :: CONST_PK //
        Set return_type to ReturnType Const :: CONST_PK so that
        the value of the auto-increment primary key column
        is returned .
    ]
];
$response = $otsClient->deleteRow($request);

```

Example 2

The following code provides an example of how to set deletion conditions:

```

$request = [
    'table_name' => ' MyTable ',
    'condition' => [
        'row_existence' => RowExistenceExpectationConst ::
CONST_EXPECT_EXIST, // Delete the row if it exists .
        'column_filter' => [ // Delete the row
            when the value of Col0 is greater than 100 .
            'column_name' => ' Col0 ',
            'value' => 100 ,
            'comparator' => Comparator TypeConst :: CONST_GREATER_THAN
        ],
    ],
    'primary_key' => [ // The primary key .
        [' PK0 ', 123 ],
        [' PK1 ', ' abc ' ]
    ]
];
$response = $otsClient->deleteRow($request);

```

8.8 Multi-row operations

The Table Store SDK provides multi-row operations, such as `BatchGetRow`, `BatchWriteRow`, and `GetRange`.

BatchGetRow

BatchGetRow

You can call this operation to read several rows of data from one or more tables.

The `BatchGetRow` operation is a set of `GetRow` operations. Each operation is executed independently, returns results, and computes the consumed capacity units (CUs) independently.

Compared to the execution of a large number of `GetRow` operations, the use of the `BatchGetRow` operation can effectively reduce the request response time and increase the data read rate.

The parameters of the BatchGetRow operation are the same as those of the GetRow operation. Note that BatchGetRow uses the same parameter conditions for all rows. For example, if columns_to_get is set to colA, only the value of colA is read for all the rows.

Similar to the use of the BatchWriteRow operation, when using the BatchGetRow operation, you must check the response. If the operation fails for some rows, the system may not throw an exception but stores information about the failed rows in BatchGetRowResponse.

Operation

```
/**
 * Read the specified rows of data .
 * Note that if the BatchGetRow operation fails
 * for some rows , the system does not throw an
 * exception but stores information about the failed
 * rows in $ response . For more information , see the
 * example of handling the return result of BatchGetRow
 * .
 * @ api
 * @ param [] $ request Request parameters .
 * @ return [] Response .
 * @ throws OTSClientException Indicates that a
 * parameter check error occurs or the server returns a
 * verification error .
 * @ throws OTSServerException Indicates that the
 * Table Store server returns an error .
 */
public function batchGetRow ( array $ request );
```

Request format

```
$ result = $ client -> batchGetRow ([
    ' tables ' => [
        REQUIRED
        [
            ' table_name ' => '< string >',
            REQUIRED
            ' primary_keys ' => [
                REQUIRED
                [
                    ['< string >', < PrimaryKey Value >],
                    ['< string >', < PrimaryKey Value >],
                    ['< string >', < PrimaryKey Value >, <
                    PrimaryKey Type >]
                ],
                // Other primary keys
            ]
        ],
        ' max_versions ' => < integer >,
        ' time_range ' => [
            ' start_time ' => < integer >,
            ' end_time ' => < integer >,
            ' specific_time ' => < integer >
        ],
    ],
],
```

```

        'start_column' => '< string >',
        'end_column' => '< string >',
        'token' => '< string >',
        'columns_to_get' => [
            '< string >',
            '< string >',
            '//...
        ],
        'column_filter' => '< ColumnCondition >'
    ],
    // Other tables
]);

```

Request format description

- Difference from GetRow
 - The `primary_key` parameter is changed to the `primary_keys` parameter, so that multiple rows can be read simultaneously.
 - Hierarchies are created for tables, so that multiple tables can be read simultaneously.
- The `tables` parameter is organized in the unit of tables, which specifies the information about the rows to be read for subsequent operations on tables.
 - `table_name`: required. This parameter specifies the name of the table.
 - `primary_keys`: required. The value is a list of primary key information of the rows. Each item in the list contains information about a primary key column. The structure of each primary key column is as follows:
 - The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
 - Each item consists of the primary key name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType, which is optional).
 - The value of PrimaryKeyValue can be an integer or a string.
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which respectively indicate the INTEGER, STRING (UTF-8

encoded string), and BINARY types. If the type is INTEGER or STRING, it can be ignored. Otherwise, the type must be specified.

- **max_versions**: the maximum number of versions from which data is read.
- **time_range**: the range of versions from which data is read. For more information, see [TimeRange](#).
 - **start_time**: the start timestamp, in milliseconds. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively. To query data within a range, specify **start_time** and **end_time**. The range is a left-closed right-open interval.
 - **end_time**: the end timestamp, in milliseconds. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively.
 - **specific_time**: the specific timestamp. To query data at a specific time, specify **specific_time**. You can set either **specific_time** or (**start_time**, **end_time**). The unit of this parameter is millisecond. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively.
- You must specify at least one of **max_versions** and **time_range**.
 - If you only specify **max_versions**, data of up to the specified number of versions is returned from the latest to the earliest.
 - If you only specify **time_range**, all data within the range is returned.
 - If you specify both **max_versions** and **time_range**, data of up to the specified number of versions within the version range is returned from the latest to the earliest.
- **columns_to_get**: the set of columns to be read. If you do not specify this parameter, all columns are read.
- **start_column**: the start column to be read, which is used for reading wide rows. The returned result includes the start column. The column names are sorted lexicographically. Assume that a table contains columns "a", "b", and "c". If the value of **start_column** is "b", the reading starts from column "b", and columns "b" and "c" are returned.
- **end_column**: the end column to be read, which is used for reading wide rows. The returned result does not include the end column. The column names are sorted lexicographically. Assume that a table contains columns "a", "b", and "c".

If the value of `end_column` is "b", the reading ends at column "b", and column "a" is returned.

- `token`: the starting position of a wide row to be read next time. This parameter is currently unavailable.
- `column_filter`: the filtering condition. Only rows meeting the condition are returned. This parameter is similar to `column_condition` in `condition`. For more information, see [Filter](#).

Result format

```
[
  ' tables ' => [
    [
      ' table_name ' => '< string >',
      ' rows ' => [
        [
          ' is_ok ' => true || false ,
          ' error ' => [
            ' code ' => '< string >',
            ' message ' => '< string >',
          ]
          ' consumed ' => [
            ' capacity_u nit ' => [
              ' read ' => < integer >,
              ' write ' => < integer >
            ]
          ],
          ' primary_key ' => [
            [< string >, < PrimaryKey Value >],
            [< string >, < PrimaryKey Value >],
            [< string >, < PrimaryKey Value >, <
PrimaryKey Type >]
          ],
          ' attribute_columns ' => [
            [< string >, < ColumnValue >, <
ColumnType >, < integer >]
            [< string >, < ColumnValue >, <
ColumnType >, < integer >]
            [< string >, < ColumnValue >, <
ColumnType >, < integer >]
          ],
          ' next_token ' => '< string >'
        ],
        // Other rows
      ]
    ],
    // Other tables
  ]
]
```

Result format description

- [BatchGetRow](#) (This section describes scenarios where reading all rows fails or reading partial rows fails. Make sure that you have read this section.)

- **MaxCompute**
- The tables parameter is organized in the unit of tables. The tables correspond to requests one by one.
 - **table_name**: the name of the table.
 - **is_ok**: indicates whether the row operation is successful. A value of true indicates that the row is read. In this case, error is invalid. A value of false indicates that the row fails to be read. In this case, consumed, primary_key, and attribute_columns are invalid.
 - **error**: the error information in the response when the operation fails.
 - **code**: the error code for the operation on the current row.
 - **message**: the error message for the operation on the current row.
 - **consumed**: the number of CUs consumed by this operation.
 - **capacity_unit**: the number of read and write CUs consumed.
 - **read**: the number of read CUs consumed.
 - **write**: the number of write CUs consumed.
 - **primary_key**: the primary key value, which is consistent with that in the request.
 - **attribute_columns**: the attribute value.
 - Each item consists of the attribute name, attribute value (ColumnValue), attribute type (ColumnType), and timestamp.
 - ColumnType can be set to ColumnTypeConst::CONST_INTEGER, ColumnTypeConst::CONST_STRING, ColumnTypeConst::CONST_BINARY, ColumnTypeConst::CONST_BOOLEAN, or ColumnTypeConst::CONST_DOUBLE, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, BOOLEAN, and DOUBLE types.
 - The timestamp is a 64-bit integer that represents the version of an attribute.
 - The attributes in the returned result are sorted by attribute name lexicographically in ascending order. The versions of the attributes are sorted by timestamp in descending order.
 - The order of the columns may be inconsistent with that of columns_to_get in the request.
 - **next_token**: the starting position of a wide row to be read next time. This parameter is currently unavailable.

Example

The following code provides an example of how to read 30 rows of data at a time:

```
// Read data from three tables , 10 rows per table
.
$ tables = array ();
for ($ i = 0 ; $ i < 3 ; $ i ++ ) {
    $ primary_ke ys = array ();
    for ($ j = 0 ; $ j < 10 ; $ j ++ ) {
        $ primary_ke ys [] = [
            [ ' pk0 ' , $ i ],
            [ ' pk1 ' , $ j ]
        ];
    }
    $ tables [] = [
        ' table_name ' => ' SampleTabl e ' . $ i ,
        ' max_versio ns ' => 1 ,
        ' primary_ke ys ' => $ primary_ke ys
    ];
}
$ request = [
    ' tables ' => $ tables
];
$ response = $ otsClient -> batchGetRo w ( $ request );

// Process each table that is returned .
foreach ( $ response [ ' tables ' ] as $ tableData ) {
    print " Handling table { $ tableData [ ' table_name ' ] } ... \n ";

    // Determine how to process each row of data in
    this table .
    foreach ( $ tableData [ ' rows ' ] as $ rowData ) {

        if ( $ rowData [ ' is_ok ' ] ) {

            // Process the data that is read .
            $ row = json_encod e ( $ rowData [ ' primary_ke y ' ] );
            print " Handling row : { $ row } \n ";

        } else {

            // Display the error informatio n .
            print " Error : { $ rowData [ ' error ' ] [ ' code ' ] } { $
rowData [ ' error ' ] [ ' message ' ] } \n ";
        }
    }
}
```

For more information, see the sample documents listed in the following table.

Document	Description
BatchGetRow1.php	Describes how to use BatchGetRow to read multiple rows from a table.
BatchGetRow2.php	Describes how to use BatchGetRow to read multiple rows from multiple tables.

Document	Description
BatchGetRow3.php	Describes how to use BatchGetRow to read multiple rows from a table with the specified columns.
BatchGetRow4.php	Describes how to use BatchGetRow to process returned results.
BatchGetRowWithColumnFilter.php	Describes how to use BatchGetRow with conditional filtering.

BatchWriteRow

[BatchWriteRow](#)

You can call this operation to insert, modify, or delete several rows of data in one or more tables.

The BatchWriteRow operation is a set of PutRow, UpdateRow, and DeleteRow operations. Each operation is executed, returns results, and computes the consumed capability units (CUs) independently.

The BatchWriteRow operation allows you to perform multiple write operations with a single request. These write operations include PutRow, UpdateRow, and DeleteRow. This operation also allows you to write data to multiple tables at one time.

The process for constructing a BatchWriteRow operation is the same as that for constructing a PutRow, UpdateRow, or DeleteRow operation. You can also set update conditions.

When calling the BatchWriteRow operation, you must check the response. During batch writing, some rows may be written while other rows may fail to be written. If the operation fails for some rows, the system may not throw an exception but stores the index and error messages of the failed rows in BatchWriteRowResponse. Some failed operations may be ignored if you do not check them. In some situations, the BatchWriteRow operation may throw an exception. For example, if the server detects that incorrect parameters exist in some operations, it may throw a parameter error exception. If an exception is thrown, none of the operations in the request is completed.

Operation

/**

```

    * Write , update , or delete the specified rows of
    data .
    * Note that if the BatchWrite Row operation fails
    for some rows , the system does not throw an
    exception but stores information about the failed
    rows in $ response . For more information , see the
    example of handling the return result of BatchWrite
    Row .
    * @ api
    * @ param [] $ request Request parameters .
    * @ return [] Response .
    * @ throws OTSClientException Indicates that a
    parameter check error occurs or the server returns a
    verification error .
    * @ throws OTSServerException Indicates that the
    Table Store server returns an error .
    */
    public function batchWrite Row ( array $ request );

```

Request format

```

$ result = $ client -> batchWrite Row ([
    ' tables ' => [
        REQUIRED
        [
            ' table_name ' => '< string >',
            REQUIRED
            ' operation_ type ' => < OperationT ype >,
            ' condition ' => [
                ' row_existe nce ' => < RowExisten ce >,
                ' column_con dition ' => < ColumnCond ition >
            ],
            ' primary_ke y ' => [
                REQUIRED
                [
                    '< string >', < PrimaryKey Value >,
                    '< string >', < PrimaryKey Value >,
                    '< string >', < PrimaryKey Value >, < PrimaryKey
                    Type >]
                ],
            ' attribute_ columns ' => [
                REQUIRED in PUT
                [
                    '< string >', < ColumnValu e >,
                    '< string >', < ColumnValu e >, < ColumnType >],
                    '< string >', < ColumnValu e >, < ColumnType >,
                    < integer >]
                ],
            ' update_of_ attribute_ columns ' => [
                REQUIRED in UPDATE
                ' PUT ' => [
                    '< string >', < ColumnValu e >,
                    '< string >', < ColumnValu e >, < ColumnType >],
                    '< string >', < ColumnValu e >, < ColumnType >,
                    < integer >]
                ],
            ' DELETE ' => [
                '< string >', < integer >],
                '< string >', < integer >],
                '< string >', < integer >],
                '< string >', < integer >]
            ],
            ' DELETE_ALL ' => [
                '< string >',

```

```

        '< string >',
        '< string >',
        '< string >'
    ],
    ],
    'return_content' => [
        'return_type' => < ReturnType >
    ]
    ],
    // Other tables
]);

```

Request format description

- This operation is a combination of PutRow, UpdateRow, and DeleteRow.
 - The operation_type parameter is added to distinguish operation types.
 - Hierarchies are created for tables, so that multiple tables can be processed simultaneously.
- The tables parameter is organized in the unit of tables, which specifies the information about the rows to be written, modified, or deleted for subsequent operations on tables.
 - table_name: required. This parameter specifies the name of the table.
 - condition: For more information, see [Conditional update](#).
 - row_existence: the row existence condition.
 - column_condition: the column-based condition.
 - operation_type: the operation type. This parameter can be set to OperationTypeConst::CONST_PUT, OperationTypeConst::CONST_UPDATE, or OperationTypeConst::CONST_DELETE, which respectively indicate the PUT, UPDATE, and DELETE types.
 - If the operation type is PUT, primary_key and attribute_columns are valid.
 - If the operation type is UPDATE, primary_key and update_of_attribute_columns are valid.
 - If the operation type is DELETE, primary_key is valid.
 - return_content: the content to be returned.
 - return_type: Currently, only this parameter is required.
 - A value of ReturnTypeConst::CONST_PK indicates that the value of the auto-increment primary key column is returned.

Result format

```
[
  ' tables ' => [
    [
      ' table_name ' => '< string >',
      ' rows ' => [
        [
          ' is_ok ' => true || false ,
          ' error ' => [
            ' code ' => '< string >',
            ' message ' => '< string >',
          ],
          ' consumed ' => [
            ' capacity_u nit ' => [
              ' read ' => < integer >,
              ' write ' => < integer >
            ],
          ],
          ' primary_key ' => [
            ['< string >', < PrimaryKey Value >],
            ['< string >', < PrimaryKey Value >],
            ['< string >', < PrimaryKey Value >, <
PrimaryKey Type >]
          ],
          ' attribute_columns ' => []
        ],
        // Other rows
      ],
      // Other tables
    ]
  ]
]
```

Result format description

- [BatchWriteRow](#)(This section describes scenarios where processing all rows fails or processing partial rows fails. Make sure that you have read this section.)

- The tables parameter is organized in the unit of tables. The tables correspond to requests one by one.
 - table_name: the name of the table.
 - is_ok: indicates whether the row operation is successful. A value of true indicates that the row is written. In this case, error is invalid. A value of false indicates that the row fails to be written.
 - error: the error information in the response when the operation fails.
 - code: the error code for the operation on the current row.
 - message: the error message for the operation on the current row.
 - consumed: the number of CUs consumed by this operation.
 - capacity_unit: the number of read and write CUs consumed.
 - read: the number of read CUs consumed.
 - write: the number of write CUs consumed.
 - primary_key: the primary key value, which is consistent with that in the request. If you set the return type to return the value of the auto-increment primary key, this primary key value is returned.
 - attribute_columns: the attribute value. It is currently empty.

Example

The following code provides an example of how to import 30 rows of data at a time:

```
// Insert data into three tables , 10 rows per
table .
$ tables = array ();
for ($ i = 0 ; $ i < 3 ; $ i ++ ) {
    $ rows = array ();
    for ($ j = 0 ; $ j < 10 ; $ j ++ ) {
        $ rows [] = [
            ' operation_ type ' => OperationT ypeConst ::
CONST_PUT ,           // The operation type is PUT .
            ' condition ' => RowExisten ceExpectat ionConst ::
CONST_IGNORE ,
            ' primary_ ke y ' => [
                [ ' pk0 ' , $ i ],
                [ ' pk1 ' , $ j ]
            ],
            ' attribute_ columns ' => [
                [ ' Col0 ' , 4 ],
                [ ' Col2 ' , ' Beijing ' ]
            ]
        ];
    }
    $ tables [] = [
        ' table_name ' => ' SampleTabl e ' . $ i ,
        ' rows ' => $ rows
    ]
}
```

```

    ];
}
$request = [
    'tables' => $tables
];
$response = $otsClient->batchWriteRow($request);
// Process each table that is returned.
foreach ($response['tables'] as $tableData) {
    print "Handling table {$tableData['table_name']} ...\n";

    // Process the result returned by PutRow in this table.
    $putRows = $tableData['rows'];

    foreach ($putRows as $rowData) {
        if ($rowData['is_ok']) {
            // Data is successfully written.
            print "Capacity Unit Consumed : {$rowData['consumed']}[" .
                'capacity_unit' . ']' . 'write' . "]\n";
        } else {
            // Display the error information.
            print "Error : {$rowData['error']}[" .
                'code' . ']' . ' {$rowData['error']}[" .
                'message' . ']\n';
        }
    }
}
}

```

For more information, see the sample documents listed in the following table.

Document	Description
BatchWriteRow1.php	Describes how to perform multiple PUT operations in BatchWriteRow.
BatchWriteRow2.php	Describes how to perform multiple UPDATE operations in BatchWriteRow.
BatchWriteRow3.php	Describes how to perform multiple DELETE operations in BatchWriteRow.
BatchWriteRow4.php	Describes how to simultaneously perform the UPDATE, PUT, and DELETE operations in BatchWriteRow.
BatchWriteRowWithColumnFilter.php	Describes how to use BatchWriteRow with conditional update.

GetRange

GetRange

You can call this operation to read data within the specified primary key range.

This operation is used to read data within a range. In Table Store tables, all rows are sorted by primary key. The primary key of a table is sequentially composed of all primary key columns. Therefore, do not assume that the rows are sorted by a certain primary key column.

The GetRange operation allows you to read data in a forward or backward direction based on a given range. You can also limit the number of rows to be read. If the range is large and the number of scanned rows or the volume of data exceeds the limit, the scan stops and the read rows and the next primary key are returned. If not all rows are read, you can initiate a request to start from where the last operation left off and read the remaining rows based on the next primary key returned by the previous operation.

Operation

```
/**
 * Read data between the start primary key and
 * the end primary key .
 * Note that the server may truncate this range .
 * You need to determine whether to call the GetRange
 * operation again based on next_start _primary_k ey in
 * the response .
 * You can specify the maximum number of rows to
 * be read .
 * When specifying the start and end primary keys
 * , you can use INF_MIN and INF_MAX to represent the
 * minimum and maximum values . For more informatio n , see
 * the following code example .
 * @ api
 * @ param [] $ request Request parameters .
 * @ return [] Response .
 * @ throws OTSClientE xception Indicates that a
 * parameter check error occurs or the server returns a
 * verificati on error .
 * @ throws OTSServerE xception Indicates that the
 * Table Store server returns an error .
 */
public function getRange ( array $ request );
```

Request format

```
$ result = $ client -> getRange ([
    ' table_name ' => '< string >',
    // REQUIRED
    ' inclusive_ start_prim ary_key ' => [
        // REQUIRED
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >, < PrimaryKey Type >]
```

```

    ],
    'exclusive_end_primary_key' => [
        // REQUIRED
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >, < PrimaryKey Type >]
    ],
    'direction' => < Direction >,
    // REQUIRED
    'limit' => < Direction >,
    'max_versions' => < integer >,
    'time_range' => [
        'start_time' => < integer >,
        'end_time' => < integer >,
        'specific_time' => < integer >
    ],
    'start_column' => '< string >',
    'end_column' => '< string >',
    'token' => '< string >',
    'columns_to_get' => [
        '< string >',
        '< string >',
        //...
    ],
    'column_filter' => < ColumnCondition >
]);

```

Request format description

- Difference from GetRow
 - The `primary_key` parameter is changed to the `inclusive_start_primary_key` and `exclusive_end_primary_key` parameters. The range is a left-closed right-open interval.
 - The `direction` parameter is added to indicate the direction.
 - The `limit` parameter is added to limit the number of returned rows.
- `table_name`: required. This parameter specifies the name of the table.
- `inclusive_start_primary_key`: the primary key from which the read starts. If the specified row exists, it is included in the response. This parameter is required.
 - The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the

- primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
- Each item consists of the primary key name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType, which is optional).
 - The value of PrimaryKeyValue can be an integer or a string.
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), and BINARY types. If the type is INTEGER or STRING, it can be ignored. Otherwise, the type must be specified.
 - A primary key column can also be of the INF_MIN or INF_MAX type. INF_MIN columns are always smaller than other columns, while INF_MAX columns are always larger than other columns.
 - If PrimaryKeyType is set to PrimaryKeyTypeConst::CONST_INF_MIN or PrimaryKeyTypeConst::CONST_INF_MAX, you can set PrimaryKeyValue to null.
- **exclusive_end_primary_key**: the primary key at which the read ends. No matter whether the specified row exists, it is excluded from the response. This parameter is required.
- The primary key of a table can include multiple primary key columns. The primary key columns are sorted by the order they are added. For example, the primary key structure PRIMARY KEY (A, B, C) is different from PRIMARY KEY (A, C, B). Table Store sorts rows based on the values of all primary key columns.
 - Each item consists of the primary key name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType, which is optional).
 - The value of PrimaryKeyValue can be an integer or a string.
 - PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which respectively indicate the INTEGER, STRING (UTF-8 encoded string

), and BINARY types. If the type is INTEGER or STRING, it can be ignored.

Otherwise, the type must be specified.

- A primary key column can also be of the INF_MIN or INF_MAX type. INF_MIN columns are always smaller than other columns, while INF_MAX columns are always larger than other columns.
- If PrimaryKeyType is set to PrimaryKeyTypeConst::CONST_INF_MIN or PrimaryKeyTypeConst::CONST_INF_MAX, you can set PrimaryKeyValue to null.
- **direction**: required. This parameter specifies the order of this query. If direction is set to DirectionConst::CONST_FORWARD (the forward direction), the value of inclusive_start_primary must be smaller than that of exclusive_end_primary. All rows in the response are sorted by primary key in ascending order. If direction is set to DirectionConst::CONST_BACKWARD (the backward direction), the value of inclusive_start_primary must be greater than that of exclusive_end_primary. All rows in the response are sorted by primary key in descending order.
- **limit**: the maximum number of rows to be returned.
 - If the number of rows queried exceeds this value, the response includes a breakpoint to record the position where the read ends in this operation, so that the next read can start from this position. This value must be greater than 0.
 - Whether or not this value is set, Table Store returns a maximum of 5,000 rows of data, and the total data size does not exceed 4 MB.
- **max_versions**: the maximum number of versions from which data is read.
- **time_range**: the range of versions from which data is read. For more information, see [TimeRange](#).
 - **start_time**: the start timestamp, in milliseconds. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively. To query data within a range, specify start_time and end_time. The range is a left-closed right-open interval.
 - **end_time**: the end timestamp, in milliseconds. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively.
 - **specific_time**: the specific timestamp. To query data at a specific time, specify specific_time. You can set either specific_time or [start_time, end_time). The unit of this parameter is millisecond. The minimum and maximum values of the timestamp are 0 and INT64. MAX, respectively.

- You must specify at least one of `max_versions` and `time_range`.
 - If you only specify `max_versions`, data of up to the specified number of versions is returned from the latest to the earliest.
 - If you only specify `time_range`, all data within the range is returned.
 - If you specify both `max_versions` and `time_range`, data of up to the specified number of versions within the version range is returned from the latest to the earliest.
- `columns_to_get`: the set of columns to be read. If you do not specify this parameter, all columns are read.
- `start_column`: the start column to be read, which is used for reading wide rows. The returned result includes the start column. The column names are sorted lexicographically. Assume that a table contains columns "a", "b", and "c". If the value of `start_column` is "b", the reading starts from column "b", and columns "b" and "c" are returned.
- `end_column`: the end column to be read, which is used for reading wide rows. The returned result does not include the end column. The column names are sorted lexicographically. Assume that a table contains columns "a", "b", and "c". If the value of `end_column` is "b", the reading ends at column "b", and column "a" is returned.
- `token`: the starting position of a wide row to be read next time. This parameter is currently unavailable.
- `column_filter`: the filtering condition. Only rows meeting the condition are returned. This parameter is similar to `column_condition` in `condition`. For more information, see [Filter](#).

Result format

```
[
  'consumed' => [
    'capacity_unit' => [
      'read' => <integer>,
      'write' => <integer>
    ]
  ],
  'next_start_primary_key' => [
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>],
    ['<string>', <PrimaryKey Value>, <PrimaryKey Type>]
  ],
  'rows' => [
    [
      'primary_key' => [
        ['<string>', <PrimaryKey Value>],
```

```

        ['< string >', < PrimaryKey Value >],
        ['< string >', < PrimaryKey Value >, < PrimaryKey
Type >]
    ],
    'attribute_columns' => [
        ['< string >', < ColumnValue >, < ColumnType >,
< integer >]
        ['< string >', < ColumnValue >, < ColumnType >,
< integer >]
        ['< string >', < ColumnValue >, < ColumnType >,
< integer >]
    ]
    ],
    // Other rows
]
]

```

Result format description

- [GetRange](#)
- **consumed:** the number of CUs consumed by this operation.
 - **capacity_unit:** the number of read and write CUs consumed.
 - **read:** the number of read CUs consumed.
 - **write:** the number of write CUs consumed.
- **rows**
 - If direction in the request is set to `DirectionConst::CONST_FORWARD`, all rows are sorted by primary key in ascending order. If direction in the request is set

to `DirectionConst::CONST_BACKWARD`, all rows are sorted by primary key in descending order.

- The `attribute_columns` parameter for each row only contains the columns specified in `columns_to_get`. The order of the columns may be inconsistent with that of `columns_to_get` in the request.
- If the specified `columns_to_get` in the request does not contain any primary key column, the primary key is within the query range. However, a row that does not contain any attribute column in `columns_to_get` is not included in the response.
- `primary_key`: the primary key value, which is consistent with that in the request. If you set the return type to return the value of the auto-increment primary key, this primary key value is returned.
- `attribute_columns`: the attribute value.
 - Each item consists of the attribute name, attribute value (`ColumnValue`), attribute type (`ColumnType`), and timestamp.
 - `ColumnType` can be set to `ColumnTypeConst::CONST_INTEGER`, `ColumnTypeConst::CONST_STRING`, `ColumnTypeConst::CONST_BINARY`, `ColumnTypeConst::CONST_BOOLEAN`, or `ColumnTypeConst::CONST_DOUBLE`, which respectively indicate the `INTEGER`, `STRING` (UTF-8 encoded string), `BINARY`, `BOOLEAN`, and `DOUBLE` types.
 - The timestamp is a 64-bit integer that represents the version of an attribute.
 - The attributes in the returned result are sorted by attribute name lexicographically in ascending order. The versions of the attributes are sorted by timestamp in descending order.
- `next_start_primary_key`: the breakpoint of this `GetRange` operation.
 - Its format is the same as that of `primary_key`.
 - If this parameter is not specified, the response for this `GetRange` operation contains all data within the request range.
 - If this parameter is specified, the response for this `GetRange` operation includes only the data in the range [`inclusive_start_primary_key`, `next_start_primary_key`). To obtain the remaining data, set `inclusive_start_primary_key` to the value of `next_start_primary_key`, and retain the value of `exclusive_end_primary_key` in the original request to perform the subsequent `GetRange` operation.

Example

The following code provides an example of how to read data within the specified range:

```
// Read data with UID within the range [ 1 , 4 ).
// The complete primary key is required for
defining the range boundary . If the queried range
does not involve the range of values in a column ,
set the column to be infinitely great or small .
$ startPK = [
    [ ' PK0 ' , 1 ],
    [ ' PK1 ' , null , PrimaryKey TypeConst :: CONST_INF_ MIN ]
// INF_MIN indicates the minimum value .
];
// The complete primary key is required for
defining the range boundary . If the queried range
does not involve the range of values in a column ,
set the column to be infinitely great or small .
$ endPK = [
    [ ' PK0 ' , 4 ],
    [ ' PK1 ' , null , PrimaryKey TypeConst :: CONST_INF_ MAX ]
// INF_MAX indicates the maximum value .
];
$request = [
    ' table_name ' => ' SampleTabl e ' ,
    ' max_versio ns ' => 1 , // Set
this parameter so that the latest version is read .
    ' direction ' => DirectionC onst :: CONST_FORW ARD , //
Query data in the forward direction .
    ' inclusive_ start_prim ary_key ' => $ startPK , // The
start primary key .
    ' exclusive_ end_primar y_key ' => $ endPK , // The
end primary key .
    ' limit ' => 10 // A
maximum of 10 rows are returned .
];
$response = $ otsClient -> getRange ( $ request );
print " Read CU Consumed : { $ response [ ' consumed ' ] [ '
capacity_u nit ' ] [ ' read ' ] } \ n " ;

foreach ( $ response [ ' rows ' ] as $ rowData ) {
    // Process each row of data .
}
```

For more information, see the sample documents listed in the following table.

Document	Description
GetRange1.php	Describes how to use GetRange.
GetRange2.php	Describes how to use GetRange to obtain the specified columns.
GetRange3.php	Describes how to use GetRange to obtain the specified number of rows.
GetRangeWithColumnFilter.php	Describes how to use GetRange with conditional filtering.

8.9 Search index

Use IndexSort

Table Store sorts matched data based on the value of the IndexSort field in search index-based queries by default. Indexes of the nested type do not support IndexSort. The default value of the IndexSort field is PrimaryKeySort. You can specify the IndexSort field when creating an index. IndexSort determines the default order of the data that Table Store returns in search index-based queries. If you do not specify the IndexSort field, Table Store returns the query result that is sorted by the primary key.



Note:

IndexSort does not support ScoreSort or GeoDistanceSort.

Example

```
" index_sort " => array (
    array (
        ' field_sort ' => array (
            ' field_name ' => ' keyword ',
            ' order ' => SortOrderConstant::SORT_ORDER_ASC ,
            ' mode ' => SortModeConstant::SORT_MODE_AVG ,
        )
    ),
    array (
        ' pk_sort ' => array (
            ' order ' => SortOrderConstant::SORT_ORDER_ASC
        )
    ),
)
```

Specify a sorting method for queries

You can specify a sorting method for each query. Search index-based queries support the following sorting methods. You can also specify multiple sorting methods with different priorities as needed. The query result is sorted by these sorting methods in sequence based on their priorities from high to low.

ScoreSort

You can use ScoreSort to sort the query result by keyword relevance score. ScoreSort is applicable to scenarios such as full-text indexing. Note that you must set ScoreSort to sort the query result by keyword relevance score. Otherwise, the query result is sorted based on the value of the IndexSort field. Set the sort field as follows:

```
' sort ' => array (
    array (
        ' score_sort ' => array (
```

```

        ' order ' => SortOrderConst::SORT_ORDER_DESC
    ),
)

```

PrimaryKeySort

You can use PrimaryKeySort to sort the query result by the primary key. Set the sort field as follows:

```

' sort ' => array (
    array (
        ' pk_sort ' => array (
            ' order ' => SortOrderConst::SORT_ORDER_ASC
        )
    ),
)

```

FieldSort

You can use FieldSort to sort the query result by a specified column. Set the sort field as follows:

```

' sort ' => array (
    array (
        ' field_sort ' => array (
            ' field_name ' => ' keyword ',
            ' order ' => SortOrderConst::SORT_ORDER_ASC ,
            ' mode ' => SortModeConst::SORT_MODE_AVG ,
        )
    ),
)

```

GeoDistanceSort

You can use GeoDistanceSort to sort the query result by geographical location. Set the sort field as follows:

```

' sort ' => array (
    array (
        ' geo_distance_sort ' => array (
            ' field_name ' => ' geo ',
            ' order ' => SortOrderConst::SORT_ORDER_ASC ,
            ' distance_type ' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE ,
            ' points ' => array ( ' 0.6 ' , ' 0.6 ' )
        )
    ),
)

```

Combination of multiple sorting methods

You can sort the query result by multiple columns in sequence. Set the sort field as follows:

```
' sort ' => array (
    array (
        ' field_sort ' => array (
            ' field_name ' => ' keyword ',
            ' order ' => SortOrderC onst :: SORT_ORDER _ASC ,
            ' mode ' => SortModeC onst :: SORT_MODE_ AVG ,
        )
    ),
    array (
        ' pk_sort ' => array (
            ' order ' => SortOrderC onst :: SORT_ORDER _ASC
        )
    ),
)
```

Specify the paging method

Only fields with the EnableSortAndAgg attribute set to true can be sorted. You can set multiple filtering conditions for index sorting.

Use the limit and offset parameters

When the total number of rows to be obtained is less than 2,000, you can specify the limit and offset parameters for paging. The sum of the limit and offset parameter values cannot exceed 2,000.

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' search_que ry ' => array (
        ' offset ' => 10 ,
        ' limit ' => 10 ,
        ' get_total_ count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeC onst :: MATCH_ALL_ QUERY
        ),
        ' sort ' => array (
            array (
                ' field_sort ' => array (
                    ' field_name ' => ' keyword ',
                    ' order ' => SortOrderC onst :: SORT_ORDER
                _ASC
            )
        ),
        ' token ' => null ,
    ),
    ' columns_to _get ' => array (
        ' return_ty p e ' => ColumnRetu rnTypeCons t :: RETURN_SPE
        CIFIED ,
        ' return_nam es ' => array ( ' col1 ', ' col2 ' )
    )
);
```

```
$ response = $ otsClient -> search ($ request );
```

Use a token

If Table Store does not complete reading all required data, Table Store returns `NextToken`, which can be used to continue reading the subsequent data. When you use a token for paging, the sorting method is the same as that used in the previous request. The sorting method can be the default sorting method specified by the `IndexSort` field or a sorting method that you have specified. Additionally, you cannot set the offset parameter if you use a token. In this case, Table Store scans data page by page and cannot skip any page.

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' search_que ry ' => array (
        ' offset ' => 0 ,
        ' limit ' => 10 ,
        ' get_total_ count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeC onst :: FUNCTION_S
CORE_QUERY ,
            ' query ' => array (
                ' query ' => array (
                    ' query_type ' => QueryTypeC onst :: TERM_QUERY
                ,
                    ' query ' => array (
                        ' field_name ' => ' keyword ',
                        ' term ' => ' keyword '
                    )
                ),
                ' field_valu e_factor ' => array (
                    ' field_name ' => ' long '
                )
            ),
            ' sort ' => array (
                array (
                    ' score_sort ' => array (
                        ' order ' => SortOrderC onst :: SORT_ORDER
DESC
                    )
                ),
            ),
        ),
        ' columns_to _get ' => array (
            ' return_typ e ' => ColumnRetu rnTypeCons t :: RETURN_SPE
CIFIED ,
            ' return_nam es ' => array ( ' keyword ', ' long ' )
        )
    );

$ response = $ otsClient -> search ($ request );
print " total_hits : " . $ response [' total_hits ' ] . "\ n ";
print json_encod e ($ response [' rows ' ], JSON_PRETT Y_PRINT );

while ($ response [' next_token ' ] != null ) {
```

```

    $ request [' search_que ry '][' token ' ] = $ response ['
next_token '];
    $ request [' search_que ry '][' sort ' ] = null ;// If you
    set next_token , you do not need to specify the
    sorting method because the token includes the sorting
    informatio n .
    $ response = $ otsClient -> search ($ request );
    print json_encod e ($ response [' rows '], JSON_PRETT
Y_PRINT );
}

```

8.9.1 Query by exact match

TermQuery

You can call the TermQuery operation to query data that exactly matches the specified value of a field. When you query a Text string, Table Store tokenizes the string and exactly matches any of the tokens.

For example, Table Store tokenizes text string "tablestore is cool" into "tablestore", "is", and "cool." When you specify any of these tokens as a query string, you can retrieve the query result that contains the token.

Example

```

$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' search_que ry ' => array (
        ' offset ' => 0 ,
        ' limit ' => 2 ,
        ' get_total_ count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeC onst :: TERM_QUERY ,
            ' query ' => array (
                ' field_name ' => ' keyword ',
                ' term ' => ' keyword '
            )
        ),
        ' sort ' => array (
            array (
                ' field_sort ' => array (
                    ' field_name ' => ' keyword ',
                    ' order ' => SortOrderC onst :: SORT_ORDER
_ASC
                )
            ),
        ),
    ),
    ' columns_to _get ' => array (
        ' return_typ e ' => ColumnRetu rnTypeCons t :: RETURN_ALL
    ,
        ' return_nam es ' => array (' keyword ', ' long ')
    )
);

```

```
$ response = $ otsClient -> search ($ request );
```

TermsQuery

You can call the TermsQuery operation to query data that exactly matches the specified field values. TermsQuery is similar to TermQuery. The difference is that TermsQuery supports multiple terms. You can retrieve query results that match any of these terms.

Example

```
$ request = array (
    ' table_name ' => ' php_sdk_test ',
    ' index_name ' => ' php_sdk_test_search_index ',
    ' search_query ' => array (
        ' offset ' => 0 ,
        ' limit ' => 5 ,
        ' get_total_count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeConstant::TERMS_QUERY ,
            ' query ' => array (
                ' field_name ' => ' keyword ',
                ' terms ' => array (
                    " keyword ",
                    " php "
                )
            )
        )
    ),
    ' sort ' => array (
        array (
            ' field_sort ' => array (
                ' field_name ' => ' long ',
                ' order ' => SortOrderConstant::SORT_ORDER
                _DESC ,
                ' mode ' => SortModeConstant::SORT_MODE_AVG
            )
        )
    ),
    ' columns_to_get ' => array (
        ' return_type ' => ColumnReturnTypeConstant::RETURN_SPECIFIED ,
        ' return_names ' => array (' keyword ', ' long ')
    )
);
$ response = $ otsClient -> search ($ request );
```

8.9.2 Query by match

MatchAllQuery

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

Example

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' search_que ry ' => array (
        ' offset ' => 0 ,
        ' limit ' => 10 ,
        ' get_total_ count ' => true ,
        ' collapse ' => array (
            ' field_name ' => ' keyword '
        ),
        ' query ' => array (
            ' query_type ' => QueryTypeC onst :: MATCH_ALL_ QUERY
        ),
        ' sort ' => array ( // Set a specific sorting
// method if required .
//         array (
//             ' field_sort ' => array (
//                 ' field_name ' => ' keyword ',
//                 ' order ' => SortOrderC onst :: SORT_ORDER
// _ASC
//             )
//         ),
//     ),
    ' token ' => null ,
),
' columns_to _get ' => array (
    ' return_typ e ' => ColumnRetu rnTypeCons t :: RETURN_SPE
    CIFIED ,
    ' return_nam es ' => array ( ' col1 ', ' col2 ' )
)
);
$ response = $ otsClient -> search ( $ request );
```

MatchQuery

You can call the MatchQuery operation to query a table based on approximate matches. For example, if you search for the phrase "this is", you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is."

Example

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' search_que ry ' => array (
        ' offset ' => 0 ,
        ' limit ' => 2 ,
        ' get_total_ count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeC onst :: MATCH_QUER Y ,
            ' query ' => array (
                ' field_name ' => ' text ',
                ' text ' => ' ots text php keyword ',
//             ' operator ' => QueryOpera torConst :: PBAND ,
                ' operator ' => QueryOpera torConst :: PBOR , // Use
                minimum_sh ould_match with OR .
            )
        )
    )
);
```

```

        ' minimum_should_match ' => 3
    ),
    'sort ' => array (
        array (
            ' field_sort ' => array (
                ' field_name ' => ' keyword ',
                ' order ' => SortOrderConstant :: SORT_ORDER
                _ASC
            ),
        ),
    ),
    'columns_to_get ' => array (
        ' return_type ' => ColumnReturnTypeConstant :: RETURN_SPECIFIED ,
        ' return_names ' => array ( ' text ' )
    )
);
$response = $otsClient -> search ($ request );

```

MatchPhraseQuery

You can call the MatchPhraseQuery operation to search matched phrases.

MatchPhraseQuery is similar to MatchQuery. The difference is that MatchPhraseQuery matches a phrase as a whole. For example, if you search for the phrase "this is", you can obtain query results such as "..., this is tablestore" and "this is a table." You cannot obtain query results such as "this table is ..." and "is this a table."

Example

```

$ request = array (
    ' table_name ' => ' php_sdk_test ',
    ' index_name ' => ' php_sdk_test_search_index ',
    ' search_query ' => array (
        ' offset ' => 0 ,
        ' limit ' => 2 ,
        ' get_total_count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeConstant :: MATCH_PHRASE_QUERY ,
            ' query ' => array (
                ' field_name ' => ' text ',
                ' text ' => ' text keyword '
            )
        ),
        ' sort ' => array (
            array (
                ' field_sort ' => array (
                    ' field_name ' => ' keyword ',
                    ' order ' => SortOrderConstant :: SORT_ORDER
                    _ASC
                ),
            ),
        ),
    ),
    'columns_to_get ' => array (
        ' return_type ' => ColumnReturnTypeConstant :: RETURN_SPECIFIED ,
    )
);

```

```

        'return_names' => array('text')
    );
    $response = $otsClient->search($request);

```

8.9.3 Query by prefix

PrefixQuery

You can call the PrefixQuery operation to query data that matches a specified prefix. For a table that contains a text string, Table Store tokenizes the string and matches any of the tokens with the specified prefix.

Example

```

$request = array (
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array (
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array (
            'query_type' => QueryTypeConstant::PREFIX_QUERY,
            'query' => array (
                'field_name' => 'keyword',
                'prefix' => 'key'
            )
        ),
        'sort' => array (
            array (
                'field_sort' => array (
                    'field_name' => 'keyword',
                    'order' => SortOrderConstant::SORT_ORDER_ASC
                )
            )
        ),
        'columns_to_get' => array (
            'return_type' => ColumnReturnTypeConstant::RETURN_ALL,
            'return_names' => array('keyword')
        )
    );
$response = $otsClient->search($request);

```

8.9.4 Query by range

RangeQuery

You can call the RangeQuery operation to query data that falls within a specified range. For a table that contains a text string, Table Store tokenizes the string and matches any of the tokens that falls within the specified range.

Example

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' search_que ry ' => array (
        ' offset ' => 0 ,
        ' limit ' => 2 ,
        ' get_total_ count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeC onst :: RANGE_QUE R Y ,
            ' query ' => array (
                ' field_name ' => ' long ',
                ' range_from ' => 100 ,
                ' include_lo wer ' => true ,
                ' range_to ' => 101 ,
                ' include_up per ' => false
            )
        ),
        ' sort ' => array (
            array (
                ' field_sort ' => array (
                    ' field_name ' => ' keyword ',
                    ' order ' => SortOrderC onst :: SORT_ORDER
                )
            ),
            array (
                ' field_sort ' => array (
                    ' field_name ' => ' keyword ',
                    ' order ' => SortOrderC onst :: SORT_ORDER
                )
            )
        ),
        ' columns_to _get ' => array (
            ' return_typ e ' => ColumnRetu rnTypeCons t :: RETURN_SPE
            CIFIED ,
            ' return_nam es ' => array ( ' double ', ' long ', ' keyword
            ' )
        )
    );
$response = $ otsClient -> search ($ request );
```

8.9.5 Query by wildcard character

WildcardQuery

You can call the WildcardQuery operation to match wildcard characters in a query. You can specify a value to be matched as a string with one or more wildcard characters.

An asterisk (*) represents a number of characters or an empty string. A question mark (?) represents any single character. For example, if you search for string "table*e", you can retrieve query results such as "tablestore." The string specified as a filtering condition cannot start with an asterisk (*).

Example

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
```

```

        'search_query' => array (
            'offset' => 0,
            'limit' => 2,
            'get_total_count' => true,
            'query' => array (
                'query_type' => QueryTypeC  onst :: WILDCARD_QUERY,
                'query' => array (
                    'field_name' => 'keyword',
                    'value' => 'key *'
                )
            ),
            'sort' => array (
                array (
                    'field_sort' => array (
                        'field_name' => 'keyword',
                        'order' => SortOrderC  onst :: SORT_ORDER
_ASC
                    ),
                ),
            ),
        ),
        'columns_to_get' => array (
            'return_type' => ColumnReturnTypeCons  t :: RETURN_SPECIFIED,
            'return_names' => array ('keyword')
        )
    );
    $response = $otsClient->search($request);

```

8.9.6 Query by geographic location

Search index-based queries support three geographic location-based queries: GeoBoundingBoxQuery, GeoDistanceQuery, and GeoPolygonQuery.

GeoBoundingBoxQuery

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

Example

```

$request = array (
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array (
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array (
            'query_type' => QueryTypeC  onst :: GEO_BOUNDING_BOX_QUERY,
            'query' => array (
                'field_name' => 'geo',
                'top_left' => '31, 119',
                'bottom_right' => '29, 121'
            )
        )
    )

```

```

    ),
    'sort' => array (
        array (
            'geo_distance_sort' => array (// Sorting by
distance .
                'field_name' => 'geo ',
                'order' => SortOrderC onst :: SORT_ORDER _ASC
            ,
                'distance_type' => GeoDistanceTypeConst ::
GEO_DISTANCE_PLANE ,
                'points' => array (' 30 , 120 ')
            )
        ),
    ),
    'columns_to_get' => array (
        'return_type' => ColumnReturnTypeConst :: RETURN_SPE
CIFIED ,
        'return_names' => array (' geo ')
    )
);
$response = $otsClient -> search ($ request );

```

GeoDistanceQuery

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

Example

```

$request = array (
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array (
        'offset' => 0 ,
        'limit' => 2 ,
        'get_total_count' => true ,
        'query' => array (
            'query_type' => QueryTypeConst :: GEO_DISTANCE_
CE_QUERY ,
            'query' => array (
                'field_name' => 'geo ',
                'center_point' => ' 30 . 001 , 120 . 001 ',
                'distance' => 1000 // Unit : meters .
            )
        ),
    ),
    'sort' => array (
        array (
            'geo_distance_sort' => array (
                'field_name' => 'geo ',
                'order' => SortOrderC onst :: SORT_ORDER _ASC
            ,
                'distance_type' => GeoDistanceTypeConst ::
GEO_DISTANCE_PLANE ,
                'points' => array (' 30 , 120 ')
            )
        ),
    ),
);

```

```

    ),
    'columns_to_get' => array (
        'return_type' => ColumnReturnType::RETURN_SPECIFIED,
        'return_names' => array (' geo ')
    )
);
$response = $otsClient->search($request);

```

GeoPolygonQuery

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

Example

```

$request = array (
    'table_name' => ' php_sdk_test ',
    'index_name' => ' php_sdk_test_search_index ',
    'search_query' => array (
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array (
            'query_type' => QueryTypeConstant::GEO_POLYGON_QUERY,
            'query' => array (
                'field_name' => ' geo ',
                'points' => array (// The polygon vertices .
                    " 31 , 120 ",
                    " 29 , 121 ",
                    " 29 , 119 "
                )
            )
        ),
        'sort' => array (
            array (
                'geo_distance_sort' => array (
                    'field_name' => ' geo ',
                    'order' => SortOrderConstant::SORT_ORDER_ASC,
                    'distance_type' => GeoDistanceTypeConstant::GEO_DISTANCE_PLANE,
                    'points' => array (' 30 , 120 ')
                )
            )
        ),
        'columns_to_get' => array (
            'return_type' => ColumnReturnType::RETURN_SPECIFIED,
            'return_names' => array (' geo ')
        )
    );

```

```
$ response = $ otsClient -> search ($ request );
```

8.9.7 Query by column existence

ExistsQuery

You can call the ExistsQuery operation to query rows that contain a value other than null in a specified column. (The return result does not include the rows that contain the null value in the specified column.)

Example

```
$ request = array (
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array (
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array (
            'query_type' => QueryTypeConstant::EXISTS_QUERY,
            'query' => array (
                'field_name' => 'text'
            )
        )
    ),
    'columns_to_get' => array (
        'return_type' => ColumnReturnTypeConstant::RETURN_SPECIFIED,
        'return_names' => array ('keyword', 'long', 'array')
    )
);
$response = $ otsClient -> search ($ request );
```

Special processing for the nested type

```
$ request = array (
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array (
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array (
            'query_type' => QueryTypeConstant::NESTED_QUERY,
            'query' => array (
                'path' => "nested",
                'query' => array (
                    'query_type' => QueryTypeConstant::EXISTS_QUERY,
                    'query' => array (
                        'field_name' => 'nested.nested_long',
                    )
                ),
                'score_mode' => ScoreModeConstant::SCORE_MODE_AVG
            )
        )
    )
);
```

```

    ),
    'columns_to_get' => array (
        'return_type' => ColumnReturnType::RETURN_SPECIFIED,
        'return_names' => array ('nested')
    )
);
$response = $this->otsClient->search($request);

```

8.9.8 Query by a combination of filtering conditions

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

You can combine subqueries in different ways. If you specify subqueries as `must_queries`, Table Store returns the result that matches all the subqueries. If you specify subqueries as `must_not_queries`, Table Store returns the result that does not match any of the subqueries.

```

/**
 * Use the BoolQuery operation to query data based on
 * a combination of filtering conditions. The combination
 * methods are as follows:
 * must_queries: (A AND B) Conditions 1 and 2 must
 * be met at the same time.
 * should_queries: (A OR B) At least condition 1 or
 * condition 2 must be met.
 * filter_queries: (A AND B) The document must
 * completely match all subfilters. filter_queries are
 * similar to must_queries. The difference is that the
 * scores are not calculated in filter_queries.
 * must_not_queries: (! A AND ! B) Neither condition 1
 * nor condition 2 is met.
 * minimum_should_match: The minimum number of
 * should_queries that must be met. The default value
 * is 1.
 */

```

`must_queries (A AND B)`

Example

```

$request = array (
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array (
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array (
            'query_type' => QueryTypeConst::BOOL_QUERY,
            'query' => array (
                'must_queries' => array (
                    array (
                        'query_type' => QueryTypeConst::
TERM_QUERY,

```

```

        ' query ' => array (
            ' field_name ' => ' keyword ',
            ' term ' => ' keyword '
        )
    ),
    array (
        ' query_type ' => QueryTypeC_onst ::
RANGE_QUERY ,
        ' query ' => array (
            ' field_name ' => ' long ',
            ' range_from ' => 100 ,
            ' include_lower ' => true ,
            ' range_to ' => 101 ,
            ' include_upper ' => false
        )
    )
),
),
'sort ' => array (
    array (
        ' field_sort ' => array (
            ' field_name ' => ' keyword ',
            ' order ' => SortOrderC_onst :: SORT_ORDER_ASC
        )
    ),
),
),
'columns_to_get ' => array (
    ' return_type ' => ColumnReturnTypeCons_t :: RETURN_SPECIFIED ,
    ' return_names ' => array ( ' keyword ', ' long ' )
)
);
$response = $otsClient -> search ($ request );

```

must_not_queries (! A AND ! B)

Example

```

$ request = array (
    ' table_name ' => ' php_sdk_test ',
    ' index_name ' => ' php_sdk_test_search_index ',
    ' search_query ' => array (
        ' offset ' => 0 ,
        ' limit ' => 2 ,
        ' get_total_count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeC_onst :: BOOL_QUERY ,
            ' query ' => array (
                ' must_not_queries ' => array (
                    array (
                        ' query_type ' => QueryTypeC_onst ::
TERM_QUERY ,
                        ' query ' => array (
                            ' field_name ' => ' keyword ',
                            ' term ' => ' keyword '
                        )
                    ),
                    array (
                        ' query_type ' => QueryTypeC_onst ::
RANGE_QUERY ,

```

```

        ' query ' => array (
            ' field_name ' => ' long ',
            ' range_from ' => 100 ,
            ' include_lower ' => true ,
            ' range_to ' => 101 ,
            ' include_upper ' => false
        )
    ),
),
),
),
'sort ' => array (
    array (
        ' field_sort ' => array (
            ' field_name ' => ' keyword ',
            ' order ' => SortOrderConstant::SORT_ORDER_ASC
        )
    ),
),
),
),
' columns_to_get ' => array (
    ' return_type ' => ColumnReturnType::RETURN_SPECIFIED ,
    ' return_names ' => array ( ' keyword ', ' long ' )
)
);
$response = $otsClient->search($request);

```

filter_queries (A AND B)

Example

```

$request = array (
    ' table_name ' => ' php_sdk_test ',
    ' index_name ' => ' php_sdk_test_search_index ',
    ' search_query ' => array (
        ' offset ' => 0 ,
        ' limit ' => 2 ,
        ' get_total_count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeConstant::BOOL_QUERY ,
            ' query ' => array (
                ' filter_queries ' => array (
                    array (
                        ' query_type ' => QueryTypeConstant::
TERM_QUERY ,
                        ' query ' => array (
                            ' field_name ' => ' keyword ',
                            ' term ' => ' keyword '
                        )
                    ),
                    array (
                        ' query_type ' => QueryTypeConstant::
RANGE_QUERY ,
                        ' query ' => array (
                            ' field_name ' => ' long ',
                            ' range_from ' => 100 ,
                            ' include_lower ' => true ,
                            ' range_to ' => 101 ,
                            ' include_upper ' => false
                        )
                    )
                )
            )
        )
    )
);

```

```

    ),
    ),
    ),
    'sort' => array (
        array (
            'field_sort' => array (
                'field_name' => 'keyword ',
                'order' => SortOrderC onst :: SORT_ORDER
                _ASC
            )
        )
    ),
    ),
    'columns_to_get' => array (
        'return_type' => ColumnRetu rnTypeCons t :: RETURN_SPE
        CIFIED ;
        'return_names' => array ('keyword ', 'long ')
    )
);
$response = $otsClient -> search ($request );

```

should_queries (A OR B)

Example

```

$request = array (
    'table_name' => 'php_sdk_test ',
    'index_name' => 'php_sdk_test_search_index ',
    'search_query' => array (
        'offset' => 0 ,
        'limit' => 2 ,
        'get_total_count' => true ,
        'query' => array (
            'query_type' => QueryTypeC onst :: BOOL_QUERY ,
            'query' => array (
                'should_queries' => array (
                    array (
                        'query_type' => QueryTypeC onst ::
                        TERM_QUERY ,
                        'query' => array (
                            'field_name' => 'keyword ',
                            'term' => 'keyword '
                        )
                    ),
                    array (
                        'query_type' => QueryTypeC onst ::
                        RANGE_QUERY ,
                        'query' => array (
                            'field_name' => 'long ',
                            'range_from' => 100 ,
                            'include_lower' => true ,
                            'range_to' => 101 ,
                            'include_upper' => false
                        )
                    )
                )
            ),
            'minimum_should_match' => 1
        )
    ),
    'sort' => array (
        array (
            'field_sort' => array (
                'field_name' => 'keyword ',

```

```

        ' order ' => SortOrderC  onst :: SORT_ORDER
        _ASC
    ),
    ),
    ),
    ' columns_to _get ' => array (
        ' return_type ' => ColumnReturnTypeInfoC  onst :: RETURN_SPE
        CIFIED ;
        ' return_names ' => array ( ' keyword ', ' long ')
    )
);
$response = $otsClient -> search ($ request );

```

8.9.9 Query by nested field

You can call the NestedQuery operation to query data based on a nested field. You must use the NestedQuery operation to wrap the nested field before using the nested field to query data. In NestedQuery, you must specify a subquery of any type and the path of the nested field.

Example

```

$request = array (
    ' table_name ' => ' php_sdk_test ',
    ' index_name ' => ' php_sdk_test_search_index ',
    ' search_query ' => array (
        ' offset ' => 0 ,
        ' limit ' => 2 ,
        ' get_total_count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeInfoC  onst :: NESTED_QUERY ,
            ' score_mode ' => ScoreModeC  onst :: SCORE_MODE_AVG ,
            ' query ' => array (
                ' path ' => " nested ",
                ' query ' => array (
                    ' query_type ' => QueryTypeInfoC  onst :: TERM_QUERY
                ,
                    ' query ' => array (
                        ' field_name ' => ' nested . nested_keyword '
                    ,
                        ' term ' => ' sub '
                    )
                )
            )
        ),
        ' sort ' => array (
            array (
                ' field_sort ' => array (
                    ' field_name ' => ' nested . nested_long ',
                    ' order ' => SortOrderC  onst :: SORT_ORDER
                _DESC ,
                ' nested_filter ' => array (
                    ' path ' => " nested ",
                    ' query ' => array (
                        ' query_type ' => QueryTypeInfoC  onst ::
                        TERM_QUERY ,
                        ' query ' => array (
                            ' field_name ' => ' nested .
                            nested_keyword ',

```

```

        'term' => 'sub'
    )
    )
    ),
    ),
    ),
    ),
    'columns_to_get' => array (
        'return_type' => ColumnReturnType::RETURN_SPECIFIED,
        'return_names' => array ('nested')
    )
);
$response = $otsClient->search($request);

```

8.9.10 Sorting and paging

Use IndexSort

Table Store sorts matched data based on the value of the IndexSort field in search index-based queries by default. Indexes of the nested type do not support IndexSort. The default value of the IndexSort field is PrimaryKeySort. You can specify the IndexSort field when creating an index. IndexSort determines the default order of the data that Table Store returns in search index-based queries. If you do not specify the IndexSort field, Table Store returns the query result that is sorted by the primary key.



Note:

IndexSort does not support ScoreSort or GeoDistanceSort.

Example

```

"index_sort" => array (
    array (
        'field_sort' => array (
            'field_name' => 'keyword',
            'order' => SortOrderConstant::SORT_ORDER_ASC,
            'mode' => SortModeConstant::SORT_MODE_AVG,
        )
    ),
    array (
        'pk_sort' => array (
            'order' => SortOrderConstant::SORT_ORDER_ASC
        )
    ),
)

```

Specify a sorting method for queries

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

different priorities as needed. The query result is sorted by these sorting methods in sequence based on their priorities from high to low.

ScoreSort

You can use ScoreSort to sort the query result by keyword relevance score. ScoreSort is applicable to scenarios such as full-text indexing. Note that you must set ScoreSort to sort the query result by keyword relevance score. Otherwise, the query result is sorted based on the value of the IndexSort field. Set the sort field as follows:

```
' sort ' => array (
    array (
        ' score_sort ' => array (
            ' order ' => SortOrderC  onst :: SORT_ORDER  _DESC
        )
    ),
)
```

PrimaryKeySort

You can use PrimaryKeySort to sort the query result by the primary key. Set the sort field as follows:

```
' sort ' => array (
    array (
        ' pk_sort ' => array (
            ' order ' => SortOrderC  onst :: SORT_ORDER  _ASC
        )
    ),
)
```

FieldSort

You can use FieldSort to sort the query result by a specified column. Set the sort field as follows:

```
' sort ' => array (
    array (
        ' field_sort ' => array (
            ' field_name ' => ' keyword ',
            ' order ' => SortOrderC  onst :: SORT_ORDER  _ASC ,
            ' mode ' => SortModeCo  nst :: SORT_MODE_  AVG ,
        )
    ),
)
```

GeoDistanceSort

You can use GeoDistanceSort to sort the query result by geographical location. Set the sort field as follows:

```
' sort ' => array (
```

```

        array (
            'geo_distan' => array (
                'field_name' => 'geo',
                'order' => SortOrderC onst :: SORT_ORDER _ASC ,
                'distance_t ype' => GeoDistanc eTypeConst ::
GEO_DISTAN CE_PLANE ,
                'points' => array (' 0 . 6 , 0 . 6 ')
            )
        ),
    )
)

```

Combination of multiple sorting methods

You can sort the query result by multiple columns in sequence. Set the sort field as follows:

```

' sort ' => array (
    array (
        'field_sort' => array (
            'field_name' => 'keyword',
            'order' => SortOrderC onst :: SORT_ORDER _ASC ,
            'mode' => SortModeCo nst :: SORT_MODE_ AVG ,
        )
    ),
    array (
        'pk_sort' => array (
            'order' => SortOrderC onst :: SORT_ORDER _ASC
        )
    ),
)

```

Specify the paging method

Only fields with the EnableSortAndAgg attribute set to true can be sorted. You can set multiple filtering conditions for index sorting.

Use the limit and offset parameters

When the total number of rows to be obtained is less than 2,000, you can specify the limit and offset parameters for paging. The sum of the limit and offset parameter values cannot exceed 2,000.

```

$ request = array (
    'table_name' => 'php_sdk_te st',
    'index_name' => 'php_sdk_te st_search_ index',
    'search_que ry' => array (
        'offset' => 10 ,
        'limit' => 10 ,
        'get_total_ count' => true ,
        'query' => array (
            'query_type' => QueryTypeC onst :: MATCH_ALL_ QUERY
        ),
        'sort' => array (
            array (
                'field_sort' => array (
                    'field_name' => 'keyword',

```

```

        ' order ' => SortOrderC  onst :: SORT_ORDER
_ASC
    ),
    ),
    ' token ' => null ,
),
' columns_to _get ' => array (
    ' return_type ' => ColumnReturnTypesCons t :: RETURN_SPE
CIFIED ,
    ' return_names ' => array ( ' col1 ', ' col2 ' )
)
);
$response = $otsClient -> search ($ request );

```

Use a token

If Table Store does not complete reading all required data, Table Store returns `NextToken`, which can be used to continue reading the subsequent data. When you use a token for paging, the sorting method is the same as that used in the previous request. The sorting method can be the default sorting method specified by the `IndexSort` field or a sorting method that you have specified. Additionally, you cannot set the `offset` parameter if you use a token. In this case, Table Store scans data page by page and cannot skip any page.

```
$ request = array (
    ' table_name ' => ' php_sdk_te st ',
    ' index_name ' => ' php_sdk_te st_search_ index ',
    ' search_que ry ' => array (
        ' offset ' => 0 ,
        ' limit ' => 10 ,
        ' get_total_ count ' => true ,
        ' query ' => array (
            ' query_type ' => QueryTypeC onst :: FUNCTION_S
CORE_QUERY ,
            ' query ' => array (
                ' query ' => array (
                    ' query_type ' => QueryTypeC onst :: TERM_QUERY
,
                    ' query ' => array (
                        ' field_name ' => ' keyword ',
                        ' term ' => ' keyword '
                    )
                ),
                ' field_valu e_factor ' => array (
                    ' field_name ' => ' long '
                )
            )
        ),
        ' sort ' => array (
            array (
                ' score_sort ' => array (
                    ' order ' => SortOrderC onst :: SORT_ORDER
_DESC
                )
            ),
        ),
    ),
),
```

```

        ' columns_to_get ' => array (
            ' return_type ' => ColumnReturnType::RETURN_SPECIFIED ;
            ' return_names ' => array ( ' keyword ', ' long ')
        )
    );

    $ response = $ otsClient -> search ($ request );
    print " total_hits : " . $ response [' total_hits '] . "\ n ";
    print json_encode ($ response [' rows '], JSON_PRETTY_PRINT );

    while ($ response [' next_token '] != null ) {
        $ request [' search_query '][' token '] = $ response ['
next_token '];
        $ request [' search_query '][' sort '] = null ;// If you
        set next_token , you do not need to specify the
        sorting method because the token includes the sorting
        information .
        $ response = $ otsClient -> search ($ request );
        print json_encode ($ response [' rows '], JSON_PRETTY
        PRINT );
    }

```

8.10 Conditional update

The conditional update feature updates table data only when the specified conditions are met. If the conditions are not met, the update fails. Conditional update is applicable to the condition parameter of the PutRow, UpdateRow, DeleteRow, and BatchWriteRow operations.

Judgment conditions are divided into row existence conditions and column-based conditions.

- The row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST, which are respectively expressed by RowExistenceExpectationConst::CONST_IGNORE, RowExistenceExpectationConst::CONST_EXPECT_EXIST, and RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST. When a table needs to be updated, the system first checks the row existence conditions. If a row existence condition is not met, the row is not updated and the system throws an exception.
- Column-based conditions currently include SingleColumnValueCondition and CompositeColumnValueCondition. They are used to make condition-based judgments based on the values of one or more columns, similar to the conditions used by Table Store filters.

Conditional update can be used to implement optimistic locking. When a row needs to be updated, the system first obtains the value of a column. Assume that column A

has a value of 1. Set the condition "Column A = 1", update the row, and set column A to 2. If the update fails, the row has been updated by another client.

Format

```
' condition ' => [
    ' row_existence ' => < RowExistenceExpectation >
    ' column_condition ' => < ColumnCondition >
]
```

The structure of SingleColumnValueCondition is as follows:

```
[
    ' column_name ' => '< string >',
    ' value ' => < ColumnValue >,
    ' comparator ' => < ComparatorType >
    ' pass_if_missing ' => true || false
    ' latest_version_only ' => true || false
]
```

The structure of CompositeColumnValueCondition is as follows:

```
[
    ' logical_operator ' => < LogicalOperator >
    ' sub_conditions ' => [
        < ColumnCondition >,
        < ColumnCondition >,
        < ColumnCondition >,
        // Other conditions
    ]
]
```

Alternatively, when only a row existence condition exists (in most cases), the format can be as follows:

```
' condition ' => < RowExistenceExpectation >
```

Format description

- **row_existence**: the row existence condition.
 - The row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST,
 - which are respectively expressed by RowExistenceExpectationConst::CONST_IGNORE, RowExistenceExpectationConst::CONST_EXPECT_EXIST, and RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST.
 - When a table needs to be updated, the system first checks the row existence conditions. If a row existence condition is not met, the row is not updated and the system throws an exception.
 - **column_condition**: the column-based condition.
 - SingleColumnValueCondition supports comparison between the value of a column (which can be a primary key column) and a constant. It does not support comparison between two columns or between two constants.
- **column_name**: the column name.
 - **value**: the column value.
 - The column value is in the format of [Value, Type]. Type can be set to ColumnTypeConst::CONST_INTEGER, ColumnTypeConst::CONST_STRING, ColumnTypeConst::CONST_BINARY, ColumnTypeConst::CONST_BOOLEAN, or ColumnTypeConst::CONST_DOUBLE, which respectively indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, BOOLEAN, and

DOUBLE types. If the type is BINARY, it must be specified. Otherwise, the type can be ignored.

- If the type is not BINARY, the column value can be in the format of [Value].
- **comparator:** [ComparatorType](#).
 - **EQUAL:** expressed by `ComparatorTypeConst::CONST_EQUAL`.
 - **NOT_EQUAL:** expressed by `ComparatorTypeConst::CONST_NOT_EQUAL`.
 - **GREATER_THAN:** expressed by `ComparatorTypeConst::CONST_GREATER_THAN`.
 - **GREATER_EQUAL:** expressed by `ComparatorTypeConst::CONST_GREATER_EQUAL`.
 - **LESS_THAN:** expressed by `ComparatorTypeConst::CONST_LESS_THAN`.
 - **LESS_EQUAL:** expressed by `ComparatorTypeConst::CONST_LESS_EQUAL`.
- **pass_if_missing:** Attribute columns of a row in Table Store are not fixed. A row may not contain the column involved in comparison. This parameter specifies whether such rows are returned.
 - A value of true indicates that a row is returned if the specified column does not exist in the row.
 - A value of false indicates that a row is not returned if the specified column does not exist in the row.
 - The default value is true.
- **latest_version_only:** indicates whether the condition is valid only for the latest version.
 - A value of true indicates that a row is returned if the value of the latest version for the specified column in the row meets the condition.
 - A value of false indicates that a row is returned if the value of any version for the specified column in the row meets the condition.
 - The default value is true.
- **CompositeColumnValueCondition** has a tree structure. It contains the inner node **logical_operator** and the leaf node **SingleColumnValueCondition**.
 - **logical_operator:** [LogicalOperator](#), in the enumeration type.
 - **NOT:** expressed by `LogicalOperatorConst::CONST_NOT`.
 - **AND:** expressed by `LogicalOperatorConst::CONST_AND`.
 - **OR:** expressed by `LogicalOperatorConst::CONST_OR`.

- **sub_conditions:** the sub-node, which can be a `SingleColumnValueCondition` or `CompositeColumnValueCondition`.
- Two or more sub-nodes can be mounted on the AND or OR operator. Only one sub-node can be mounted on the NOT operator.

Example 1

The following code provides an example of how to construct a `SingleColumnValueCondition`:

```
// Set the filter so that a row is returned if
the value of Col0 is 0 .
$column_condition = [
    'column_name' => 'Col0 ',
    'value' => 0 ,
    'comparator' => Comparator::CONST_EQUAL
    'pass_if_missing' => false //
    If a row does not contain the Col0 column, the
    row is not returned .
    'latest_version_only' => true //
    Only the values of the latest version is checked .
];
```

Example 2

The following code provides an example of how to construct a `CompositeColumnValueCondition`:

```
// The condition composite1 is ( Col0 = 0 ) AND ( Col1 > 100 ).
$composite1 = [
    'logical_operator' => LogicalOperator::CONST_AND
    ,
    'sub_conditions' => [
        [
            'column_name' => 'Col0 ',
            'value' => 0 ,
            'comparator' => Comparator::CONST_EQUAL
        ],
        [
            'column_name' => 'Col1 ',
            'value' => 100 ,
            'comparator' => Comparator::CONST_GREATER_THAN
        ]
    ]
];
// The condition composite2 is ( ( Col0 = 0 ) AND ( Col1 > 100 ) ) OR ( Col2 <= 10 ).
$composite2 = [
    'logical_operator' => LogicalOperator::CONST_OR
    ,
    'sub_conditions' => [
        $composite1 ,
    ]
];
```

```

        [
            'column_name' => 'Col2 ',
            'value' => 10,
            'comparator' => Comparator::TypeConst::
CONST_LESS_EQUAL
        ]
    ];

```

Example 3

You can use conditional update to implement optimistic locking. The following code provides an example of how to use conditional update to update a column with an incremented value.

```

// Read a row .
$request = [
    'table_name' => 'MyTable ',
    'primary_key' => [ // The primary key .
        ['PK0 ', 123 ],
        ['PK1 ', 'abc ' ]
    ],
    'max_versions' => 1
];
$response = $otsClient->getRow ($ request );
$columnMap = getColumnValueAsMap ($ response ['attribute_
columns ']);
$col0Value = $ columnMap [' col0 '][ 0 ][ 1 ];
// Configure conditional update for Col0 to
increment the value of Col0 by 1 .
$request = [
    'table_name' => 'MyTable ',
    'condition' => [
        'row_exists' => RowExistenceExpectationConst
::CONST_EXPECT_EXIST,
        'column_condition' => [ // If the
condition is met, the data is updated .
            'column_name' => ' col0 ',
            'value' => $ col0Value ,
            'comparator' => Comparator::TypeConst::
CONST_EQUAL
        ]
    ],
    'primary_key' => [ // The primary key .
        ['PK0 ', 123 ],
        ['PK1 ', 'abc ' ]
    ],
    'update_of_attribute_columns' => [
        'PUT' => [
            [' col0 ', $ col0Value + 1 ]
        ]
    ]
];

```

```
$ response = $ otsClient -> updateRow ($ request );
```

8.11 Filter

Table Store filters are used to filter results that the server reads so that the server only returns the rows or columns matching the filtering conditions. Filters are applicable to the `column_filter` parameter of the `GetRow`, `BatchGetRow`, and `GetRange` operations.

Currently, Table Store only supports `SingleColumnValueFilter` and `CompositeColumnValueFilter`, which are used to determine whether to filter out the data of a row based on reference column values. `SingleColumnValueFilter` only checks the value of a single reference column, whereas `CompositeColumnValueFilter` checks the values of multiple reference columns and forms a logical combination of the check results to determine whether to filter out the data of a row.

Note that the filters are used to filter the data that has been read. Therefore, the reference columns used by `SingleColumnValueFilter` or `CompositeColumnValueFilter` must be included in the read data. If you specify the columns from which data is read but these columns do not include any reference columns, the filters cannot obtain reference column values. When a reference column does not exist, `SingleColumnValueFilter` uses the `pass_if_missing` parameter to determine whether the filtering conditions are met. That is, you can select an action even when no reference column exists.

Format

```
' column_filter ' => < ColumnFilter >
```

The structure of `SingleColumnValueFilter` is as follows:

```
[
    ' column_name ' => '< string >',
    ' value ' => < ColumnValue >,
    ' comparator ' => < Comparator Type >
    ' pass_if_missing ' => true || false
    ' latest_version_only ' => true || false
]
```

The structure of `CompositeColumnValueFilter` is as follows:

```
[
    ' logical_operator ' => < LogicalOperator >
    ' sub_filters ' => [
```

```
        < ColumnFilter > ,  
        < ColumnFilter > ,  
        < ColumnFilter > ,  
        // Other conditions  
    ]  
]
```

Format description

- **column_filter**: the filtering condition.
 - **SingleColumnValueFilter** supports comparison between the value of a column (which can be a primary key column) and a constant. It does not support comparison between two columns or between two constants.
- **column_name**: the column name.
- **value**: the column value.
 - The column value is in the format of [Value, Type]. Type can be set to `ColumnTypeConst::CONST_INTEGER`, `ColumnTypeConst::CONST_STRING`, `ColumnTypeConst::CONST_BINARY`, `ColumnTypeConst::CONST_BOOLEAN`, or `ColumnTypeConst::CONST_DOUBLE`, which respectively indicate the `INTEGER`, `STRING` (UTF-8 encoded string), `BINARY`, `BOOLEAN`, and

DOUBLE types. If the type is BINARY, it must be specified. Otherwise, the type can be ignored.

- If the type is not BINARY, the column value can be in the format of [Value].
- **comparator:** [ComparatorType](#).
 - **EQUAL:** expressed by `ComparatorTypeConst::CONST_EQUAL`.
 - **NOT_EQUAL:** expressed by `ComparatorTypeConst::CONST_NOT_EQUAL`.
 - **GREATER_THAN:** expressed by `ComparatorTypeConst::CONST_GREATER_THAN`.
 - **GREATER_EQUAL:** expressed by `ComparatorTypeConst::CONST_GREATER_EQUAL`.
 - **LESS_THAN:** expressed by `ComparatorTypeConst::CONST_LESS_THAN`.
 - **LESS_EQUAL:** expressed by `ComparatorTypeConst::CONST_LESS_EQUAL`.
- **pass_if_missing:** Attribute columns of a row in Table Store are not fixed. A row may not contain the column involved in comparison. This parameter specifies whether such rows are returned.
 - A value of true indicates that a row is returned if the specified column does not exist in the row.
 - A value of false indicates that a row is not returned if the specified column does not exist in the row.
 - The default value is true.
- **latest_version_only:** indicates whether the condition is valid only for the latest version.
 - A value of true indicates that a row is returned if the value of the latest version for the specified column in the row meets the condition.
 - A value of false indicates that a row is returned if the value of any version for the specified column in the row meets the condition.
 - The default value is true.
- **CompositeColumnValueFilter** has a tree structure. It contains the inner node **logical_operator** and the leaf node **SingleColumnValueFilter**.
 - **logical_operator:** [LogicalOperator](#), in the enumeration type.
 - **NOT:** expressed by `LogicalOperatorConst::CONST_NOT`.
 - **AND:** expressed by `LogicalOperatorConst::CONST_AND`.
 - **OR:** expressed by `LogicalOperatorConst::CONST_OR`.

- **sub_filters:** the sub-node, which can be a `SingleColumnValueFilter` or `CompositeColumnValueFilter`.
- Two or more sub-nodes can be mounted on the AND or OR operator. Only one sub-node can be mounted on the NOT operator.

Example 1

The following code provides an example of how to construct a `SingleColumnValueFilter`:

```
// Set the filter so that a row is returned
if the value of Col0 is 0 .
$column_filter = [
    'column_name' => 'Col0 ',
    'value' => 0 ,
    'comparator' => Comparator::CONST_EQUAL,
    'pass_if_missing' => false //
    If a row does not contain the Col0 column, the
    row is not returned .
    'latest_version_only' => true //
    Only the values of the latest version is checked .
];
```

Example 2

The following code provides an example of how to construct a `CompositeColumnValueFilter`:

```
// The condition composite1 is ( Col0 = 0 ) AND (
Col1 > 100 ).
$composite1 = [
    'logical_operator' => LogicalOperator::
CONST_AND ,
    'sub_filters' => [
        [
            'column_name' => 'Col0 ',
            'value' => 0 ,
            'comparator' => Comparator::
CONST_EQUAL,
        ],
        [
            'column_name' => 'Col1 ',
            'value' => 100 ,
            'comparator' => Comparator::
CONST_GREATER_THAN
        ]
    ]
];
// The condition composite2 is ( ( Col0 = 0 ) AND (
Col1 > 100 ) ) OR ( Col2 <= 10 ).
$composite2 = [
    'logical_operator' => LogicalOperator::
CONST_OR ,
    'sub_filters' => [
        $composite1 ,
```

```

        [
            'column_name' => 'Col2',
            'value' => 10,
            'comparator' => Comparator::TypeConst::
CONST_LESS_EQUAL
        ]
    ];

```

8.12 Auto-increment primary key column

Auto increment of primary key columns is a new feature launched by Table Store, and is available for PHP SDK V4.0.0 and later.

This feature allows you to specify a primary key column as an auto-increment column. Table Store automatically generates a new value in this column when you write data to a table. The generated value is the largest value in the column under the same partition key. This feature mainly applies to system design scenarios that require auto-increment primary key columns, such as item IDs on e-commerce websites, user IDs of large websites, post IDs in forums, and message IDs in chat tools.

Features

- Table Store currently supports multiple primary key columns. The first primary key column is the partition key and cannot be set as an auto-increment column.
- Except for the partition key, any of other primary key columns can be set as an auto-increment column.
- Auto increment of a primary key column is implemented at the partition key level because the value is increased based on the partition key.
- Only one primary key column can be set as an auto-increment column in each table.
- The automatically generated values in an auto-increment column are 64-bit signed long integers.

Operations

Auto increment of primary key columns applies to two types of operations: one to create tables and the other to write data.

- Create a table

When creating a table, you only need to set `PrimaryKeyOption` to `PrimaryKeyOptionConst::CONST_PK_AUTO_INCR` for the auto-increment primary key column.

Related operation: `CreateTable`.

```
function createTable ($otsClient)
{
    $request = [
        'table_meta' => [
            'table_name' => 'table_name',           // Specify
            the table name.
            'primary_key_schema' => [
                ['PK_1', PrimaryKeyOptionConst::CONST_STRING], // The name and type of the first primary
                key column (partition key) are PK_1 and STRING,
                respectively.
                ['PK_2', PrimaryKeyOptionConst::CONST_INTEGER, PrimaryKeyOptionConst::CONST_PK_AUTO_INCR] // The name and type of the second
                primary key column are PK_2 and INTEGER, respectively. Set this column as an auto-increment primary
                key column.
            ],
            'reserved_throughput' => [
                'capacity_unit' => [                // The reserved
                read / write throughput: 0 read CUs and 0 write CUs
                'read' => 0,
                'write' => 0
                ],
            ],
            'table_options' => [
                'time_to_live' => -1,                // The table
                never expires.
                'max_versions' => 1,                // Only one
                version is saved.
                'deviation_cell_version_in_sec' => 86400 // The valid data version offset, in seconds.
            ],
        ];
    $otsClient->createTable ($request);
}
```

- The first primary key column is the partition key and cannot be set as an auto-increment column.
- Only INTEGER columns can be set as auto-increment columns.
- Only one auto-increment primary key column is allowed in each table.

- Write data

When writing data, you only need to set `PrimaryKeyType` to `PrimaryKeyTypeConst::CONST_PK_AUTO_INCR`. The value serves as a placeholder.

Related operations: `PutRow`, `UpdateRow`, and `BatchWriteRow`.

```
function putRow ($ otsClient )
{
    $ row = [
        ' table_name ' => ' table_name ',
        ' primary_key ' => [
            [ ' PK_1 ', ' Hangzhou ' ],          // A
list of primary key names and values .
            [ ' PK_2 ', null , PrimaryKey TypeConst :: CONST_PK_A
UTO_INCR ] // The auto - increment primary key column
. You do not need to specify the value because
it is automatica lly generated by Table Store .
You only need to enter the placeholde r PrimaryKey
TypeConst :: CONST_PK_A UTO_INCR .
        ],
        ' attribute_columns ' => [              // A list of
attribute columns .
            [ ' name ', ' John ' ],              // The
attribute name , attribute value , attribute type , and
timestamp . Ignore the parameters that are not
specified .
            [ ' age ', 20 ],
            [ ' address ', ' Alibaba ' ],
            [ ' product ', ' OTS ' ],
            [ ' married ', false ]
        ],
        ' return_content ' => [
            ' return_type ' => ReturnType Const :: CONST_PK
// Set return_type to ReturnType Const :: CONST_PK so
that the value of the auto - increment primary key
column is returned .
        ]
    ];
    $ ret = $ otsClient -> putRow ( $ row );
    print_r ( $ ret );

    $ primaryKey = $ ret [ ' primary_key ' ]; //
The obtained primary key value can be used by
operations such as GetRow , UpdateRow , and DeleteRow .
    return $ primaryKey ;
}
```

- When writing data, you only need to add a placeholder to the auto-increment primary key column.
- To obtain the value of the auto-increment primary key column that is automatically generated after the data is written to Table Store, set `return_type` to `ReturnTypeConst::CONST_PK`.

8.13 Error handling

The Table Store PHP SDK adopts the `Exception` method to handle errors. If the called operation does not throw an exception, the operation succeeds. Otherwise, the operation fails.



Note:

Batch operations such as `BatchGetRow` and `BatchWriteRow` can be called only after the system verifies that no exception is thrown and the status of each row is successful.

Exception

The Table Store PHP SDK has two types of exceptions: `OTSClientException` and `OTSServerException`. Both are inherited from the `Exception` class.

- `OTSClientException` indicates an internal SDK exception, for example, incorrect parameter values.
- `OTSServerException` indicates a server error. A server error message provides details about the cause of an exception. `OTSServerException` has the following components:
 - `getHttpStatus()`: the returned HTTP status code, for example, 200 or 404.
 - `getOTSErrorCode()`: the error type string returned by Table Store.
 - `getOTSErrorMessage()`: the detailed error description returned by Table Store.
 - `getRequestId()`: the UUID that uniquely identifies a request. If the problem persists, send the request ID to Table Store development engineers for help.

8.14 FAQ

If your problem persists, contact our engineer team by the following means.

Contact us

- Go to the [Yunqi Community](#) and submit your problem in the Q&A module.
- Submit a ticket in the [ticket system](#).

- Scan the following QR code on DingTalk to join the Table Store chat group and discuss your problem with us.



9 Download SDK

9.1 Java SDK

The SDK of version 4.0.0 and later provides TTL and Max Versions, and is therefore incompatible with the SDK in Version 2.x.x.

SDK version: 4.10.2

Release date: 2019-03-11

Download: [tablestore-4.10.2](#)

Updates:

- Remove log4j2 implement and log4j2.xml.
- TunnelWorker: fix auto retry logic when network condition is too bad.

SDK version: 4.7.4

Release date: 2018-09-27

Download: [tablestore-4.7.4-release.zip](#)

Updates:

- Added SearchIndex:
 - Multi-Field Search
 - Range query
 - Wildcard search
 - Nested query
 - Full-text search
 - Ranking
- Added global secondary index.

SDK version: 4.1.0

Release date: 11/10/2016

Download: [aliyun_tablestore_java_sdk_4.1.0.zip](#)

Updates: The split points of partitions can be get from reponse of DescribeTable.

SDK version: 4.0.0

Release date: 01/08/2016

Download: [aliyun_tablestore_java_sdk_4.0.0.zip](#)

Updates:

- Provides [Time To Live](#).
- Provides [Max Versions](#).

SDK version: 2.2.4

Release date: 12/05/2016

Download: [aliyun_tablestore_java_sdk_2.2.4.zip](#)

Updates:

- Adds API condition update.
- Adds filter.

SDK version 2.1.0

Release date: 12/11/2015

Download: [aliyun-ots-java-sdk-2.1.0.zip](#)

Updates:

- Asynchronous network transmission and performance tuning: When the CPU usage is the same, the QPS is increased by several times.
- Flexible and easy-to-use asynchronous interfaces: Callback is introduced and Future is returned simultaneously.
- Unbundled from OSS SDK: The new version only includes code of TableStore SDK. The directory is slightly adjusted.
- Optimized retry logic: The default retry logic is optimized. An erroneous single row can be retried independently during batch operations. The retry logic custom method is clearer.
- Optimized log: Logs are recorded for each step from request sending to request receiving. Logs of slow requests are recorded. Logs of the whole chain from SDK to back-end services are recorded using TraceId.
- OTSWriter interface supporting batch data importing: This interface aims at providing easy-to-use and efficient data importing service for users.

- Other optimized functions: Toolbox functions for various data classes are enriched and interfaces for data size computing are provided.

**Note:**

Version 2.1.0 may be slightly incompatible with Version 2.0.4 due to the following limitations:

- When an old SDK is replaced with the new one, you must change the import paths of a few classes, because the packages of these data classes are adjusted. For example, the package of “ClientConfiguration” is changed from “com.aliyun.openservices” to “com.aliyun.openservices.ots” . The main reason why the package is adjusted is that Table Store SDK is unbundled from OSS SDK. It is therefore more appropriate to put the classes of data commonly used into the package of Table Store.
- When you no longer use an OTSClient instance (for example, before the program ends), call the shutdown method of OTSClient to release the thread and connection resources occupied by the OTSClient object.
- Names of some configuration items in “ClientConfiguration” are adjusted. For example, a time unit is added as a configuration item.
- The dependency between packages in the new SDK is changed. For example, “HttpAsyncClient” and “Jodatime” are used. If any problem occurs during SDK running, check whether a conflicting dependency is introduced.

SDK version 2.0.4

Release date: 25/09/2015

Download: [aliyun-ots-java-sdk-2.0.4.zip](#)

9.2 NodeJS SDK

Version: 4.0.6

Release date: 09/10/2016

Download: [aliyun-tablestore-nodejs-sdk-4.0.6.tar.gz](#)

GitHub: [v4.0.6](#)

Updates:

- Adds support on new feature `async` `await` and related sample codes.
- Fixes the error in reading some Chinese text.
- Adds support on STSToken.
- Adds support on the criterion `maxTimeDeviation` to the `UpdateTable` operation.
- Adds version, build, coverage, and license logos to Readme.

Version: 4.0.3

Release date: 27/08/2016

Download: [aliyun-tablestore-nodejs-sdk-4.0.3.tar.gz](#)

GitHub: [v4.0.3](#)

Updates:

- Unifies the version information.

Version: 4.0.1

Release date: 27/08/2017

Download: [aliyun-tablestore-nodejs-sdk-4.0.1.tar.gz](#)

GitHub: [v4.0.1](#)

Updates:

- Removes unnecessary code.
- Fixes the value when `GetRange` returns 0 piece of data.

Version: 4.0.0

Release date: 07/07/2017

Download: [aliyun-tablestore-nodejs-sdk-4.0.0.tar.gz](#)

GitHub: [v4.0.0](#)

9.3 Python SDK

Python SDK Development Kit Version 5.1.0

Release date: 07/22/2019

Download: [aliyun-tablestore-python-sdk-5.1.0.tar.gz](#)

Updates:

- Supports local transaction.

Python SDK Development Kit Version 4.3.0

Release date: 12/12/2017

Download: [aliyun-tablestore-python-sdk-4.3.0.tar.gz](#)

Updates:

- Supports Python 3 (Python 3.3, Python 3.4, Python 3.5 and Python 3.6).

Python SDK Development Kit Version 4.2.0

Release date: 12/09/2017

Download: [aliyun-tablestore-python-sdk-4.2.0.tar.gz](#)

Updates:

- Supports STS.

Python SDK Development Kit Version 4.1.0

Release date: 28/06/2017

Download: [aliyun-tablestore-python-sdk-4.1.0.tar.gz](#)

Updates:

- Supports python 2.6.

Python SDK Development Kit Version 4.0.0

Release date: 27/06/2017

Download: [aliyun-tablestore-python-sdk-4.0.0.tar.gz](#)

Updates:

- Supports multi-version.
- Supports TTL.
- Supports the auto-increment function of Primary Key columns.

Python SDK Development Kit Version 2.1.0

Release date: 15/10/2016

Download: [aliyun-ots-python-sdk-2.1.0.zip](#)

Updates:

- Provides Conditional Update and Filter.
- Fixes retry bugs.

Python SDK Development Kit Version 2.0.8

Release date: 30/03/2016

Download: [aliyun-ots-python-sdk-2.0.8.zip](#)

Updates:

- Provides HTTPS access and certificate verification.

Python SDK Development Kit Version 2.0.7

Release date: 30/12/2015

Download: [aliyun-ots-python-sdk-2.0.7.zip](#)

Updates:

- Refines the example code.

Python SDK Development Kit Version 2.0.6

Release date: 23/10/2015

Download: [aliyun-ots-python-sdk-2.0.6.zip](#)

Updates:

- Adjusts the backoff retry strategy for specified exceptions.

Python SDK Development Kit Version 2.0.5

Release date: 25/09/2015

Python SDK Development Kit Download: [ots_python_sdk-2.0.5.zip](#)

9.4 .Net SDK

Version: 3.0.0

Release date: 05/05/2016

Download: [aliyun_table_store_dotnet_sdk_3.0.0.zip](#)

Updates:

- Filters are added.
- Warnings generated during the compilation process are cleared.

- Useless dependent packages are cleared.
- Useless code is cleared.
- The template-called code is simplified.
- Invalid parameter checking is added.
- User-configured parameters are trimmed.
- The UserAgent header is added.
- Condition.IGNORE, Condition.EXPECT_EXIST, and Condition.EXPECT_NOT_EXIST are deleted.
- The DLL file name is changed from Aliyun.dll to Aliyun.TableStore.dll.
- Open source code is released to GitHub.

Version: 2.2.1

Release date: 14/04/2016

Download: [aliyun-ots-dotnet-sdk-2.2.1.zip](#)

Updates:

- When the SDK internally checks the HTTP response header, it ignores the case of letters.

Version: 2.2.0

Release date: 05/04/2016

Download: [aliyun-ots-dotnet-sdk-2.2.0.zip](#)

Updates:

- The default number of connections in the connection pool is increased from 50 to 300.
- The conditional update function is added.

Version: 2.1.0

Release date: 30/12/2015

Download: [aliyun-ots-dotnet-sdk-2.1.0.zip](#)

Updates:

- The reserved capacity unit settings in the sample code are adjusted based on the Pay-As-You-Go billing method.
- Adds a wider range of BatchWriteRow and BatchGetRow test cases.

9.5 PHP SDK

PHP SDK version 2.1.1

Release date: 14/01/2017

Download: [aliyun-tablestore-php-sdk-2.1.1.zip](#)

Updates:

- Supports 32 bit operation system.

PHP SDK version 2.1.0

Release date: 16/11/2016

Download: [aliyun-ots-php-sdk-2.1.0.zip](#)

Updates:

- Provides Conditional Update and Filter.
- Compatible with PHP version 5.5 and 5.6.

PHP SDK version 2.0.3

Release date: 18/05/2016

Download: [aliyun-ots-php-sdk-2.0.3.zip](#)

Updates:

- Removes delete table from example code.

PHP SDK version 2.0.2

Release date: 11/04/2016

Download: [aliyun-ots-php-sdk-2.0.2.zip](#)

Updates:

- Fix a bug of pb decode.

PHP SDK version 2.0.0

Release date: 22/09/2015

Download: [aliyun-ots-php-sdk-2.0.0.zip](#)

Updates:

- Supports all of Table Store APIs.

- Compatible with PHP version 5.3, 5.4, 5.5 and 5.6.
- Includes standard retry strategy.
- Uses Guzzle Http Client as network library.
- Uses composer as dependency management and engineering organization tools.
- Uses phpDocumentor 2 to generate programming document in HTML format.