ALIBABA CLOUD

Alibaba Cloud

Tablestore

SDK Reference

Document Version: 20220627

Alibaba Cloud

Legal disclaimer

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

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

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

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

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

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	**Danger:** Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	**Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	**Notice:** If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	**Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.SDK overview 16
2.Java SDK 17
2.1. Overview 17
2.2. Install Tablestore SDK for Java 17
2.3. Initialization 18
2.4. Table 21
2.4.1. Overview 21
2.4.2. Create data tables 21
2.4.3. Update a table 27
2.4.4. List the names of tables 29
2.4.5. Predefined columns 29
2.4.6. Split data by a specified size 31
2.4.7. Query the description of tables 31
2.4.8. Delete a table 32
2.4.9. Configure an auto-increment primary key column 33
2.4.10. Configure conditional update 35
2.4.11. Configure local transaction 39
2.4.12. Configure atomic counter 41
2.4.13. Configure a filter 43
2.5. Single-row operations 46
2.6. Multi-row operations 57
2.7. Incremental data operations 68
2.8. Search Index 70
2.8.1. Create search indexes 70
2.8.2. List search indexes 72
2.8.3. Query the description of a search index 73

2.8.4. Delete search indexes 74
2.8.5. Term query 74
2.8.6. Terms query 76
2.8.7. Match all query 78
2.8.8. Match query 80
2.8.9. Phrase query 83
2.8.10. Prefix query 85
2.8.11. Range query 87
2.8.12. Wildcard query 87
2.8.13. Geo query 89
2.8.14. Boolean query 92
2.8.15. Nested query 98
2.8.16. Sorting and pagination 100
2.8.17. Exists query 103
2.8.18. Collapse (distinct) 105
2.8.19. Aggregation 106
2.8.20. Parallel scan 127
2.8.21. Virtual columns 136
2.9. Secondary Index 137
2.9.1. Global secondary index 137
2.9.2. Local secondary index 143
2.10. Tunnel service 149
2.10.1. Overview 149
2.10.2. Quick start 149
2.10.3. Create tunnels 155
2.10.4. List the information of a tunnel 157
2.10.5. Query the information of a tunnel 158
2.10.6. Delete tunnels 161

2.11. SQL query 162
2.11.1. Overview 162
2.11.2. Create mapping tables for tables 162
2.11.3. Delete mapping tables 163
2.11.4. List the names of tables 164
2.11.5. Query information about tables 165
2.11.6. Query the index information about tables 166
2.11.7. Query data in tables 168
2.12. Data Delivery 169
2.12.1. Create delivery tasks 169
2.12.2. List the names of delivery tasks 172
2.12.3. Query the descriptive information of delivery tasks 173
2.12.4. Delete delivery tasks 174
2.13. TimeSeries model 175
2.13.1. Overview 175
2.13.2. Create a time series table 176
2.13.3. Query the names of time series tables 176
2.13.4. Query the information about a time series table 176
2.13.5. Update a time series table 177
2.13.6. Delete a time series table 178
2.13.7. Write time series data 178
2.13.8. Query time series data 179
2.13.9. Retrieve time series 179
2.13.10. Update time series metadata 179
2.13.11. Delete time series metadata 181
2.14. Error handling 182
3.Go SDK 184
3.1. Preface 184

3.2. Installation 184
3.3. Initialization 184
3.4. Table 186
3.4.1. Create a data table 186
3.4.2. UpdateTable 193
3.4.3. List the names of tables 194
3.4.4. Predefined columns 195
3.4.5. Query the description of a table 196
3.4.6. Delete tables 197
3.4.7. Configure an auto-increment primary key column 198
3.4.8. Configure conditional update 200
3.4.9. Configure local transaction 203
3.4.10. Configure atomic counter 206
3.4.11. Configure filter 208
3.5. Single-row operations 211
3.6. Multi-row operations 218
3.7. Search Index 225
3.7.1. Create search indexes 225
3.7.2. List search indexes 229
3.7.3. Query the description of a search index 230
3.7.4. Delete search indexes 231
3.7.5. Term query 232
3.7.6. Terms query 234
3.7.7. Match all query 236
3.7.8. Match query 238
3.7.9. Phrase query 241
3.7.10. Prefix query 243
3.7.11. Range query 245

3.7.12. Wildcard query 247
3.7.13. Geo query 249
3.7.14. Boolean query 254
3.7.15. Nested query 257
3.7.16. Sorting and pagination 258
3.7.17. Exists query 263
3.7.18. Collapse (distinct) 264
3.7.19. Aggregation 266
3.7.20. Parallel scan 286
3.8. Secondary Index 290
3.8.1. Global secondary index 290
3.8.2. Local secondary index 295
3.9. Tunnel Service 300
3.9.1. Install Tunnel Service 300
3.9.2. Quick start 301
3.9.3. Configuration items 302
3.9.4. Create tunnels 304
3.9.5. Query the information of a tunnel 305
3.9.6. Delete tunnels 307
3.10. SQL query 307
3.10.1. Overview 307
3.10.2. Create mapping tables 308
3.10.3. Delete mapping tables 309
3.10.4. List the names of tables 309
3.10.5. Query information about tables 310
3.10.6. Query index information about tables 312
3.10.7. Query data 313
3.11. Data Delivery 316

3.11.1. Create delivery tasks 316
3.11.2. List the names of delivery tasks 320
3.11.3. Query the descriptive information of delivery tasks 321
3.11.4. Delete delivery tasks 322
3.12. TimeSeries model 323
3.12.1. Overview 323
3.12.2. Create a time series table 324
3.12.3. Query the names of time series tables 325
3.12.4. Query the information about a time series table 326
3.12.5. Update a time series table 327
3.12.6. Delete a time series table 328
3.12.7. Write time series data 329
3.12.8. Query time series data 330
3.12.9. Retrieve time series 331
3.12.10. Update time series metadata 333
4.Python SDK 335
4.1. Preface 335
4.2. Installation 336
4.3. Initialization 337
4.4. Table 339
4.4.1. Overview 339
4.4.2. Create data tables 339
4.4.3. UpdateTable 346
4.4.4. List the names of tables 346
4.4.5. Query the description of a table 347
4.4.6. Delete tables 348
4.4.7. Configure an auto-increment primary key column 349
4.4.8. Configure conditional update 350

4.4.9. Configure local transaction 353
4.4.10. Configure atomic counter 355
4.4.11. Filter 357
4.5. Single-row operations 359
4.6. Multi-row operations 370
4.7. Search Index 380
4.7.1. Create search indexes 381
4.7.2. Query the description of a search index 384
4.7.3. List search indexes 384
4.7.4. Delete search indexes 385
4.7.5. Term query 385
4.7.6. Terms query 387
4.7.7. Match all query 389
4.7.8. Match query 391
4.7.9. Match phrase query 393
4.7.10. Prefix query 395
4.7.11. Range query 397
4.7.12. Wildcard query 399
4.7.13. Exists query 401
4.7.14. Geo query 403
4.7.15. Boolean query 405
4.7.16. Nested query 409
4.7.17. Sorting and pagination 411
4.7.18. Aggregation 414
4.8. Secondary Index 425
4.8.1. Global secondary index 425
4.8.2. Local secondary index 427
4.9. Error handling 430

5.Node.js SDK 432
5.1. Preface 432
5.2. Installation 432
5.3. Initialization 432
5.4. Data type 434
5.5. Table 434
5.5.1. Create data tables 434
5.5.2. UpdateTable 443
5.5.3. List the names of tables 444
5.5.4. Query the description of a table 445
5.5.5. Delete tables 445
5.5.6. Configure an auto-increment primary key column 446
5.5.7. Configure conditional update 449
5.5.8. Configure local transaction 451
5.5.9. Configure atomic counter 455
5.5.10. Configure a filter 457
5.6. Single-row operations 459
5.7. Multi-row operations 467
5.8. Search Index 475
5.8.1. Create search indexes 475
5.8.2. List search indexes 479
5.8.3. Query the description of a search index 480
5.8.4. Delete search indexes 480
5.8.5. Term query 481
5.8.6. Terms query 483
5.8.7. Match all query 485
5.8.8. Match query 487
5.8.9. Phrase query 490

5.8.10. Prefix query 492
5.8.11. Range query 494
5.8.12. Wildcard query 496
5.8.13. Geo query 498
5.8.14. Boolean query 503
5.8.15. Nested query 506
5.8.16. Sorting and pagination 507
5.9. Secondary Index 512
5.9.1. Global secondary index 512
5.9.2. Local secondary index 517
5.10. Error handling 522
6..NET SDK 523
6.1. Preface 523
6.2. Installation 523
6.3. Initialization 524
6.4. Table 526
6.4.1. Create tables 526
6.4.2. UpdateTable 531
6.4.3. List the names of tables 532
6.4.4. Query the description of a table 533
6.4.5. Delete tables 534
6.4.6. Configure an auto-increment primary key column 535
6.4.7. Configure conditional update 538
6.4.8. Configure atomic counter 540
6.4.9. Configure filter 542
6.5. Single-row operations 546
6.6. Multi-row operations 558
6.7. Search Index 567

6.7.1. Create search indexes 567
6.7.2. Query the description of a search index 572
6.7.3. List search indexes 573
6.7.4. Delete search indexes 574
6.7.5. Term query 575
6.7.6. Match all query 576
6.7.7. Match query 577
6.7.8. Phrase query 580
6.7.9. Prefix query 582
6.7.10. Range query 583
6.7.11. Wildcard query 584
6.7.12. Geo query 586
6.7.13. Boolean query 589
6.7.14. Nested query 591
6.7.15. Sorting and pagination 592
6.8. Global secondary index 595
6.9. Troubleshooting 599
7.PHP SDK 601
7.1. Preface 601
7.2. Install Tablestore SDK for PHP 602
7.3. Initialization 606
7.4. Table operations 608
7.4.1. Create tables 608
7.4.2. UpdateTable 614
7.4.3. List the names of tables 618
7.4.4. Split data by a specified size 619
7.4.5. Query the description of a table 622
7.4.6. Delete tables 625

7.4.7. Configure an auto-increment primary key column 626
7.4.8. Configure conditional update 629
7.4.9. Configure local transaction 634
7.4.10. Configure filter 638
7.5. Single-row operations 642
7.6. Multi-row operations 663
7.7. Search index 682
7.7.1. Create search indexes 687
7.7.2. Delete search indexes 691
7.7.3. Query the description of a search index 692
7.7.4. List search indexes 693
7.7.5. Term query 694
7.7.6. Terms query 696
7.7.7. Match all query 698
7.7.8. Match query 700
7.7.9. Phrase query 702
7.7.10. Prefix query 704
7.7.11. Range query 706
7.7.12. Wildcard query 708
7.7.13. Geo query 710
7.7.14. Exists query 715
7.7.15. Collapse (distinct) 718
7.7.16. Boolean query 719
7.7.17. Nested query 726
7.7.18. Sorting and pagination 728
7.8. Global secondary index 733
7.9. Troubleshooting 735
8.Download SDK 737

8.1. Tablestore SDK for Java 737
8.2. Tablestore SDK for Node.js 740
8.3. Tablestore SDK for Python 741
8.4. Tablestore SDK for .NET 743
8.5. PHP SDK 745

1.SDK overview

This topic describes Tablestore SDKs for different programming languages.

Before you use Tablestore SDKs, you must:

- Understand and activate Tablestore. For more information, see the Tablestore product page.
- Create an AccessKey pair. For more information, see Create an AccessKey pair.

The following table lists the Tablestore SDKs for different programming languages.

Language	Reference
Java	Java SDK
Python	Python SDK
Go	Go SDK
.NET	.NET SDK
Node.js	Node.js SDK

2.Java SDK

2.1. Overview

This topic describes the prerequisites, version compatibility, and versions of Tablestore SDK for Java. This topic applies to Tablestore SDK for Java V5.0.0 or later.

Prerequisites

- Tablestore is activated.
- An AccessKey pair is created. For more information, see Obtain an AccessKey pair.
- Tablestore SDK for Java V4.0.0 or later supports data versions and time to live (TTL). For more information about data versions and TTL, see Data versions and TTL.

Version compatibility

Latest version: 5.13.5

- Tablestore SDK for Java V4.x.x: compatible
- Tablestore SDK for Java V2.x.x: incompatible

Versions

For more information about the versions of Tablestore SDK for Java, see Tablestore SDK for Java.

2.2. Install Tablestore SDK for Java

This topic describes how to install Tablestore SDK for Java.

Prerequisites

To install Tablestore SDK for Java, you must use JDK 6.0 or later.

Installation methods

You can install Tablestore SDK for Java by using one of the following two methods:

- Add dependencies to a Maven project

  To use Tablestore SDK for Java in Maven, you need only to add the corresponding dependencies to the pom.xml file. In this example, Tablestore SDK for Java V5.13.5 is used. Add the following content to <dependencies>.

  > **Note** For more information about version history of Tablestore SDK for Java, see Tablestore SDK for Java.

  ```
  <dependency>
      <groupId>com.aliyun.openservices</groupId>
      <artifactId>tablestore</artifactId>
      <version>5.13.5</version>
  </dependency>
  ```

- Import the JAR package into Eclipse

In this example, Tablestore SDK for Java V5.13.5 is used. To import the JAR package into Eclipse, perform the following steps:

i. Download the Tablestore SDK for Java package.
ii. Decompress the SDK for Java package.
iii. Copy the tablestore-<versionId>.jar file and all files in the lib folder to your project.

<versionId> specifies the version of Tablestore SDK for Java. For example, the file name for V5.13.5 is tablestore-5.13.5.jar.

iv. Right-click your project name in Eclipse and choose **Properties > Java Build Path > Add JARs**.
v. Select all JAR files that you copied in Step 3.

Sample programs

Tablestore SDK for Java 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.

2.3. Initialization

OTSClient is a client for Tablestore. OTSClient provides various methods that you can use to manage tables and perform read and write operations on a single row or multiple rows. Before you can use Tablestore SDK for Java to send a request, you must initialize an OTSClient instance and modify the default configurations of ClientConfiguration. Before you can use the TimeSeries model, you must initialize the TimeseriesClient.

Obtain an endpoint

An endpoint is a domain name that is used to access a Tablestore instance in a region. The following table describes the examples of endpoints in the supported formats.

Example	Description
http://sun.cn-hangzhou.ots.aliyuncs.com	The public endpoint that is used to access the sun instance in the China (Hangzhou) region over HTTP.
https://sun.cn-hangzhou.ots.aliyuncs.com	The public endpoint that is used to access the sun instance in the China (Hangzhou) region over HTTPS.

Notice You can connect to Tablestore by using public endpoints or private endpoints. For more information, see Endpoint.

To query the endpoints of your Tablestore instance, perform the following steps:

1. Log on to the Tablestore console.
2. Click the name of the instance to go to the **Instance Details** tab.

   On the Instance Details tab, you can view the endpoints of the instance in the Instance Access URL section.

Configure an AccessKey pair

Before you can connect to Tablestore, you must have an AccessKey pair that consists of an AccessKey ID and an AccessKey secret. The AccessKey pair is used to verify your identity. Tablestore supports the following types of AccessKey pairs:

- The AccessKey pair of an Alibaba Cloud account. To obtain the AccessKey pair of an Alibaba Cloud account, perform the following steps:
  i. On the Alibaba Cloud official website, register an Alibaba Cloud account.
  ii. Create an AccessKey pair in the RAM console.
- The AccessKey pair of a RAM user that is authorized to access Tablestore. To obtain the AccessKey pair of a RAM user, perform the following steps:
  i. Log on to the RAM console by using an Alibaba Cloud account. Then, create a RAM user or find an existing RAM user.
  ii. Grant the RAM user the permissions to access Tablestore.
  iii. After the RAM user is granted the permissions to access Tablestore, you can use the AccessKey pair of the RAM user to access Tablestore.
- Temporary access credentials that are obtained from Security Token Service (STS). The following process shows how temporary access credentials are obtained from STS:
  i. The application server uses RAM or STS to obtain access credentials that consist of a temporary AccessKey ID, an AccessKey secret, and a token, and then sends the access credentials to you.
  ii. You can use the access credentials to access Tablestore.

Initialize a client

If you want to use the Tablestore SDK, you must create a client and call the operations in the client to access Tablestore. The operations in the client provide the same functions as the RESTful APIs provided by Tablestore.

The Tablestore SDK provides the following clients: SyncClient and AsyncClient. SyncClient contains synchronous operations and AsyncClient contains asynchronous operations. If you call a synchronous operation, the operation directly returns a response, which indicates that the request is executed. You can call synchronous operations to obtain information about various features of Tablestore. Compared with synchronous operations, asynchronous operations are more flexible. Multithreading provides higher performance than asynchronous operations. You can choose asynchronous operations or multithreading based on your business requirements.

> **Note** SyncClient and AsyncClient are thread-safe. You can use the clients to manage internal threads and connection resources. We recommend that you create a global client. This way, you do not need to create a client for each thread or request.

After you obtain the AccessKey ID and AccessKey secret, you can perform the following steps to initialize a client:

- Use the default configurations to create a SyncClient

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

- Use the custom configurations to create a SyncClient

```
// ClientConfiguration provides multiple configuration items. Only the most commonly used
items are listed.
ClientConfiguration clientConfiguration = new ClientConfiguration();
// Set the timeout until a connection is established. Unit: milliseconds.
clientConfiguration.setConnectionTimeoutInMillisecond(5000);
// Specify a timeout period for the socket connection. Unit: milliseconds.
clientConfiguration.setSocketTimeoutInMillisecond(5000);
// Configure a retry policy. If you do not configure a retry policy, the default retry po
licy is used.
clientConfiguration.setRetryStrategy(new AlwaysRetryStrategy());
SyncClient client = new SyncClient(endPoint, accessId, accessKey, instanceName, clientCon
figuration);
```

Initialize the TimeseriesClient

If you want to use the TimeSeries model, you must initialize the TimeseriesClient.

The Tablestore SDK provides the TimeseriesClient and AsyncTimeseriesClient methods. You can use the TimeseriesClient method for synchronous operations the AsyncTimeseriesClient method for asynchronous operations.

After you obtain the AccessKey ID and AccessKey secret, you can perform the following steps to initialize the TimeseriesClient:

- Use the default configurations to create a TimeseriesClient

```
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
TimeseriesClient client = new TimeseriesClient(endPoint, accessKeyId, accessKeySecret, in
stanceName);
```

- Use the custom configurations to create a TimeseriesClient

```
// ClientConfiguration provides multiple configuration items. Only the most commonly used
items are listed.
ClientConfiguration clientConfiguration = new ClientConfiguration();
// Set the timeout until a connection is established. Unit: milliseconds.
clientConfiguration.setConnectionTimeoutInMillisecond(5000);
// Specify a timeout period for the socket connection. Unit: milliseconds.
 clientConfiguration.setSocketTimeoutInMillisecond(5000);
// Configure a retry policy. If you do not configure a retry policy, the default retry po
licy is used.
clientConfiguration.setRetryStrategy(new AlwaysRetryStrategy());
TimeseriesClient client = new TimeseriesClient(endPoint, accessId, accessKey, instanceNam
e, clientConfiguration);
```

HTTPS

Upgrade to Java 7 for connection based on HTTPS.

Multithreading

- Multithreading is supported.

- We recommend that you use the same OTSClient object to run multithreading tasks.

2.4. Table

2.4.1. Overview

Tablestore SDK for Java provides you a variety of operations to perform on tables.

Operation	Description
Create tables	Creates a table or creates a table and its global secondary indexes.
Update tables	Updates the configurations of a table.
Query the description of a table	Queries the configurations of a table.
Query table names	Queries the names of all tables in an instance.
ComputeSplitsBySize	Logically splits the data in a table into several partitions whose sizes are close to a specified size.
Delete tables	Deletes a specified table.
Create global secondary indexes	Creates global secondary indexes for a base table. After you create an index table, you can read data from the index table or delete the index table.

2.4.2. Create data tables

This topic describes how to create a data table by calling the CreateTable operation. When you call the CreateTable operation, you must specify the schema information and configuration information of the data table when you call the CreateTable operation. You can set reserved read throughput and reserved write throughput for data tables in high-performance instances. You can create one or more index tables when you create a data table.

> Note
>
> - After a data table is created, it takes several seconds to load the data table. During this period, read and write operations performed on the table may fail. Perform operations on the data table after the data table is loaded.
> - You must specify the primary key when you create a data table. A primary key can contain one to four primary key columns. Each primary key column has a name and a data type.

Prerequisites

- An instance is created in the Tablestore console. For more information, see Create instances.
- Client is initialized. For more information, see Initialization.

Parameters

Parameter	Description
tableMeta	The schema information of the data table. The schema information contains the following items: • tableName: the name of the data table. • primaryKey: the schema of the primary key for the data table. For more information, see Primary keys and attributes. **Note** Attribute columns are optional. Different rows in Tablestore can have different attribute columns. You can specify the names of attribute columns when you write data to the attribute columns. ◦ The primary key of a data table can contain one to four primary key columns. Primary key columns are sorted in the order in which they are added. For example, PRIMARY KEY (A, B, C) and PRIMARY KEY (A, C, B) have different primary key schemas. Tablestore sorts rows based on the values of all primary key columns. ◦ The first primary key column serves as the partition key. Data that has the same partition key value is stored in the same partition. Therefore, we recommend that you assign no more than 10 GB of data to partitions that have the same partition key value. Otherwise, a single partition may be too large to split. We recommend that you evenly distribute requests to read and write data to partitions that have different partition keys for load balancing. • definedColumns: the predefined columns of the table and the data types of the predefined column values. Primary key columns cannot be specified as predefined columns. You can use predefined columns as the indexed columns or attribute columns for index tables.

Parameter	Description
tableOptions	The configuration information of the data table. For more information, see Data versions and TTL. The configurations include the following items: • timeToLive: the period for which data in the data table can be retained. This period is the validity period of data. When the retention period exceeds the timeToLive value, Tablestore automatically deletes expired data. The minimum timeToLive is 86400, which is equal to one day. A value of -1 specifies that data never expires. When you create a data table, you can set timeToLive to -1 if you expect that data never expires. After the table is created, you can call the UpdateTable operation to modify the timeToLive value. Unit: seconds. **Note** You must set timeToLive to -1 if you want to create an index table for a data table. • maxVersions: the maximum number of versions of data that can be retained for a single attribute column. When the number of versions of data in attribute columns exceeds the maxVersions value, Tablestore automatically deletes the data of earlier versions. When you create a data table, you can customize the maximum number of versions that can be retained for the data in an attribute column. After the data table is created, you can call the UpdateTable operation to modify the maxVersions value for the data table. **Note** You must set maxVersions to 1 if you want to create an index table for a data table. • maxTimeDeviation: the max version offset, which is the maximum difference between the timestamp of the written data and the current system time. The difference between the version number and the data written time must be within the value of maxTimeDeviation. Otherwise, an error occurs when the data is written. Unit: seconds. The valid version range of data in an attribute column is calculated based on the formula: Valid version range = **[Data written time - Max version offset, Data written time + Max version offset)**. When you create a data table, Tablestore uses the default value of 86400 if you do not specify a max version offset. After the data table is created, you can call the UpdateTable operation to modify the max version offset value. Unit: seconds.

Parameter	Description
reservedThroughtput	The reserved read throughput and reserved write throughput of the data table. The reserved read throughput and reserved write throughput can be set to 0 for data tables only in capacity instances. Reserved throughput does not apply to these instances. The default value 0 specifies that all throughput is billed on a pay-as-you-go basis. Unit: capacity unit (CU). • If reserved read throughput or reserved write throughput is set to a value greater than 0 for a data table, Tablestore allocates and reserves related resources for the data table. After the data table is created, Tablestore charges reserved throughput resources. Additional throughput is billed on a pay-as-you-go basis. For more information, see Billing overview. • If reserved read throughput or reserved write throughput is set to 0, Tablestore does not allocate or reserve related resources for the table.

Parameter	Description
indexMetas	The schema information of the index tables. Each indexMeta contains the following items: • indexName: the name of the index table. • primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. • definedColumn: the indexed attribute columns. The attribute columns are a combination of predefined columns of the data table. • indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ◦ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to index and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ◦ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. • indexUpdateMode: the update mode of the global secondary index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ◦ If indexUpdateMode is not specified or set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the global secondary index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ◦ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the update mode to IUM_SYNC_INDEX.

Examples

- Create a data table without creating an index table

```
private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME, PrimaryKeyTy
pe.STRING)); // Specify a primary key column for the data table.
        int timeToLive = -1; // Specify the validity period of data. A value of -1 specif
ies that the data never expires. Unit: seconds. You must set timeToLive to -1 for a data
table that has index tables.
        int maxVersions = 3; // Specify the maximum number of versions that can be retain
ed for each column. A value of 1 specifies that only the latest version is retained for e
ach column. You must set maxVersions to 1 if you want to create an index table for a data
table.
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
        request.setReservedThroughput(new ReservedThroughput(new CapacityUnit(0, 0))); //
Set the reserved read throughput and reserved write throughout. The reserved read through
put and reserved write throughput can be set to 0 for data tables only in capacity instan
ces. However, you can set a non-zero value for data tables in high-performance instances.

        client.createTable(request);
}
```

- Create a data table and an index table whose index type is global secondary index

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType
.STRING)); // Add a primary key column to the data table.
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType
.INTEGER)); // Add a primary key column to the data table.
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnT
ype.STRING)); // Add a predefined column to the data table.
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnT
ype.INTEGER)); // Add a predefined column to the data table.
    int timeToLive = -1; // Specify the validity period of data. A value of -1 specifies
that the data never expires. Unit: seconds. You must set timeToLive to -1 for a data tabl
e that has index tables.
    int maxVersions = 1; // Specify the maximum number of versions that can be retained f
or each column. A value of 1 specifies that only the latest version is retained for each
column. You must set maxVersions to 1 if you want to create an index table for a data tab
le.
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME);
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // Add a primary key column to the
index table.
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); // Add an attribute column to the ind
ex table.
    indexMetas.add(indexMeta);
    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMet
as); // Create the index table when you create the data table.
    client.createTable(request);
}
```

- Create a data table and an index table whose index type is local secondary index

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType
.STRING)); // Add a primary key column to the data table.
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType
.INTEGER)); // Add a primary key column to the data table.
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnT
ype.STRING)); // Add a predefined column to the data table.
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnT
ype.INTEGER)); // Add a predefined column to the data table.
    int timeToLive = -1; // Specify the validity period of data. A value of -1 specifies
that the data never expires. Unit: seconds. You must set timeToLive to -1 for a data tabl
e that has index tables.
    int maxVersions = 1; // Specify the maximum number of versions that can be retained f
or each column. A value of 1 specifies that only the latest version is retained for each
column. You must set maxVersions to 1 if you want to create an index table for a data tab
le.
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME);
    indexMeta.setIndexType(IT_LOCAL_INDEX);     // Set the index type to IT_LOCAL_INDEX,
which specifies a local secondary index.
    indexMeta.setIndexUpdateMode(IUM_SYNC_INDEX);  // Set the index update mode to IUM_SY
NC_INDEX, which specifies the synchronous update mode. If the index type is set to IT_LOC
AL_INDEX, the index update mode must be set to IUM_SYNC_INDEX.
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1); // Add a primary key column to the
index table. The first primary key column of the index table must be the same as that of
the data table.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // Add a primary key column to the
index table.
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); // Add an attribute column to the ind
ex table.
    indexMetas.add(indexMeta);
    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMet
as); // Create the index table when you create the data table.
    client.createTable(request);
}
```

2.4.3. Update a table

You can call the UpdateTable operation to modify TableOptions and ReservedThroughput.

Parameters

- TableOptions

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

Parameter	Definition	Description

Parameter	Definition	Description
TTL	The time to live, which indicates the lifespan of data.	○ Unit: seconds. ○ If you do not want data to expire, set TTL to -1. ○ Tablestore checks whether data has expired based on the timestamp, current time, and TTL values. Data is expired and deleted if the following condition is met: `Current time - Data timestamp > Table TTL`. ○ When you write data configured with TTL, make sure that you specify a valid timestamp. We recommend that you set MaxTimeDeviation to specify the timestamp.
MaxTimeDeviation	The maximum version offset between the timestamp of written data and the current system time.	○ By default, the system generates a timestamp for new data. The system checks whether data is expired based on the timestamp. You can specify the timestamp of written data. When the version offset between the timestamp of written data and the current system time is greater than the specified TTL, the written data immediately expires. To solve this problem, 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 greater than the value of MaxVersions, Tablestore keeps only the latest versions. The number of the kept versions is equal to the MaxVersions value.

- ReservedThroughtput

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

  - The minimum interval of changing the value of ReservedThroughput is one minute.
  - Fees are charged based on the reserved read and write throughput you have specified in the ReservedThroughtput field.
  - If the ReservedThroughtput value is greater than 0, fees are charged based on the reserved throughput and duration. Tablestore charges additional reserved read and write throughput on a pay-as-you-go basis. For more information, see Billing items and pricing to avoid unexpected fees.
  - The default value 0 indicates that all throughput is billed on a pay-as-you-go basis.
  - You can set the reserved read and write throughput of capacity instances only to 0. These instances do not allow reserved throughput.

Examples

Update TTL and MaxVersions for the specified table.

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

2.4.4. List the names of tables

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

> **Note** For more information about the ListTable operation, see ListTable.

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A table is created.

Examples

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

```
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);
    }
}
```

2.4.5. Predefined columns

This topic describes how to add predefined columns to or delete predefined columns from a table. You can set predefined columns to the indexed columns or indexed attribute columns when you create a secondary index.

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A table is created.

Add predefined columns

When you create a secondary index for a base table, if the base table does not contain predefined columns or the existing predefined columns are not suitable for creating a secondary index, you can add predefined columns to the base table.

- Parameters

Parameter	Description
tableName	The name of the base table.
definedColumns	The name of the predefined columns and the data types of the predefined column values. The predefined columns cannot be primary key columns. You can set predefined columns to the indexed columns or indexed attribute columns for the secondary index. This parameter includes the following configurations: ○ name: the name of the predefined column. ○ type: the data type of the predefined column value.

- Examples

  Add the following predefined columns to the base table named sampleTable: definedColumnName01 (type: STRING), definedColumnName02 (type: INTEGER), and definedColumnName03 (type: STRING).

```
public static void addDefinedColumnRequest(SyncClient client) {
AddDefinedColumnRequest addDefinedColumnRequest = new AddDefinedColumnRequest();
addDefinedColumnRequest.setTableName("sampleTable");
addDefinedColumnRequest.addDefinedColumn("definedColumnName01",DefinedColumnType.STRING);
addDefinedColumnRequest.addDefinedColumn("definedColumnName02",DefinedColumnType.INTEGER)
;
addDefinedColumnRequest.addDefinedColumn("definedColumnName03",DefinedColumnType.STRING);
client.addDefinedColumn(addDefinedColumnRequest);
}
```

Delete predefined columns

You can delete predefined columns that you no longer use from a base table.

- Parameters

Parameter	Description
tableName	The name of the base table.
name	The name of the predefined column.

- Examples

  The following code provides an example on how to delete the predefined columns definedColumnName01 and definedColumnName02 from the base table sampleTable:

```
public static void deleteDefinedColumnRequest(SyncClient client){
DeleteDefinedColumnRequest deleteDefinedColumnRequest = new DeleteDefinedColumnRequest();
deleteDefinedColumnRequest.setTableName("sampleTable");
deleteDefinedColumnRequest.addDefinedColumn("definedColumnName01");
deleteDefinedColumnRequest.addDefinedColumn("definedColumnName02");
client.deleteDefinedColumn(deleteDefinedColumnRequest);
}
```

2.4.6. Split data by a specified size

You can call the ComputeSplitsBySize operation to logically splits the data in a table into multiple shards whose sizes are close to a specified value. The locations of the split points among these splits and the hosts where the shards reside are returned. This operation is typically used to implement plans on compute engines, such as concurrency plans.

> **Note** For more information about ComputeSplitsBySize, see ComputeSplitPointsBySize.

Parameters

Parameter	Description
tableName	The name of the base table.
splitSize	The approximate size of each shard. Unit: 100 MB.

Examples

The following code provides an example on how to logically splits the data in a table into multiple shards whose sizes are close to 200 MB:

```
private static void describeTable(SyncClient client) {
    // Splits the data in the table into multiple shards based on the size of 200 MB.
    ComputeSplitsBySizeRequest request = new ComputeSplitsBySizeRequest(TABLE_NAME, 2);
    ComputeSplitsBySizeResponse response = client.computeSplitsBySize(computeSplitsBySizeRe
quest);
    System.out.println("ConsumedCapacity="   response.getConsumedCapacity().jsonize());
    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 RangeRowQuer
yCriteria, and pass RangeRowQueryCriteria to getRange() or createRangeIterator().
        System.out.println("split.getLowerBound()="   split.getLowerBound().jsonize());
        System.out.println("split.getUpperBound()="   split.getUpperBound().jsonize());
    }
}
```

2.4.7. Query the description of tables

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

> **Note** For more information about the DescribeTable operation, see DescribeTable.

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A table is created.

Parameters

Parameter	Description
tableName	The name of the table.

Examples

```
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.getTableName());
    System.out.println("The primary key of the table:");
    for (PrimaryKeySchema primaryKeySchema : tableMeta.getPrimaryKeyList()) {
        System.out.println(primaryKeySchema);
    }
    TableOptions tableOptions = response.getTableOptions();
    System.out.println("TTL of the table:"   tableOptions.getTimeToLive());
    System.out.println("MaxVersions the table:"   tableOptions.getMaxVersions());
    ReservedThroughputDetails reservedThroughputDetails = response.getReservedThroughputDet
ails();
    System.out.println("Reserved read throughput of the table:"
              reservedThroughputDetails.getCapacityUnit().getReadCapacityUnit());
    System.out.println("Reserved write throughput of the table:"
              reservedThroughputDetails.getCapacityUnit().getWriteCapacityUnit());
}
```

2.4.8. Delete a table

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

> **Note** For more information about the DeleteTable operation, see DeleteTable.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created.
- The index tables and search indexes for the table are deleted.

Parameters

Parameter	Description
tableName	The name of the data table.

Examples

The following code provides an example on how to delete a specified table:

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

2.4.9. Configure an auto-increment primary key column

This topic describes how to configure an auto-increment primary key column. You cannot set a partition key to an auto-increment column. If you write data to a table that contains an auto-increment primary key column, you do not need to specify specific values for the auto-increment primary key column because Tablestore generates values for the auto-increment primary key column. Values generated for the auto-increment primary key column are unique and consecutive within the partition that shares the same partition key value.

> **Note** Tablestore SDK for Java 4.2.0 and later allow you to configure auto-increment primary key columns.

Prerequisites

The OTSClient instance is initialized. For more information, see Initialization.

Usage notes

1. When you create a table, you cannot set a partition key to an auto-increment primary key column.

   The data type of an auto-increment primary key column can only be set to integer. Each value generated for an auto-increment primary key column is a 64-bit signed long integer.

2. When you write data to a table, you do not need to specify specific values for the auto-increment column. Instead, you need only to set placeholder values for the auto-increment column.

   If you want to obtain the values of the auto-increment column after data is written to the table, you can set ReturnType to RT_PK.

   When you query data, you must specify the values of all primary key columns. To obtain a complete primary key value, you can set ReturnType to RT_PK in PutRow, UpdateRow, or BatchWriteRow.

Examples

When you use auto-increment primary key columns, you can call the CreateTable, PutRow, UpdateRow, or BatchWriteRow operation.

1. Create a table

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

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

2. Write data

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

```
    private static void putRow(SyncClient client, String receive_id) {
        // Create the primary key.
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder
();
        // Set the values in the first primary key column to the first four digits of m
d5(receive_id).
        primaryKeyBuilder.addPrimaryKeyColumn("PK_1", PrimaryKeyValue.fromString("Hangz
hou"));
        // The second primary key column is an auto-increment primary key column, and y
ou do not need to specify specific values for it. You need only to set placerholder val
ues for the column. Tablestore generates values for the auto-increment primary key colu
mn.
        primaryKeyBuilder.addPrimaryKeyColumn("PK_2", PrimaryKeyValue.AUTO_INCREMENT);
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        RowPutChange rowPutChange = new RowPutChange("table_name", primaryKey);
        // Set ReturnType to RT_PK to include the primary key value in the returned res
ult. By default, no primary key column value is returned if ReturnType is not set.
        rowPutChange.setReturnType(ReturnType.RT_PK);
        // Add attribute columns.
        rowPutChange.addColumn(new Column("content", ColumnValue.fromString("content"))
);
        // Write data to the table.
        PutRowResponse response = client.putRow(new PutRowRequest(rowPutChange));
        // Display the returned primary key value.
        Row returnRow = response.getRow();
        if (returnRow != null) {
            System.out.println("PrimaryKey:" + returnRow.getPrimaryKey().toString());
        }
        // Display the consumed capacity units (CUs).
        CapacityUnit  cu = response.getConsumedCapacity().getCapacityUnit();
        System.out.println("Read CapacityUnit:" + cu.getReadCapacityUnit());
        System.out.println("Write CapacityUnit:" + cu.getWriteCapacityUnit());
    }
```

2.4.10. Configure conditional update

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

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created and data is written to the table.

Usage notes

When you use the PutRow, UpdateRow, DeleteRow, or BatchWriteRow operation to update data, you can use conditional update to check row existence conditions and column-based conditions. The update is successful only when the conditions are met.

Conditional update includes row existence conditions and column-based conditions.

- Row existence conditions: include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.

When you modify a table, Tablestore first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported.

- Column-based conditions: include SingleColumnValueCondition and CompositeColumnValueCondition, which are used to determine whether the conditions are met based on the values of one or more columns.
  - SingleColumnValueCondition supports the comparison between a constant and a column. SingleColumnValueCondition does not support the comparison between two columns or two constants.
  - The inner node of CompositeColumnValueCondition is used to perform logical operations. Subconditions can be SingleColumnValueCondition or CompositeColumnValueCondition.

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

Limits

Column-based conditions for conditional update support relational operators such as =, !=, >, >=, <, and <=, and logical operators such as NOT, AND, and OR. You can specify up to 10 conditions.

Parameters

Parameter	Description
RowExistenceExpectation	When you modify a table, Tablestore first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported. Row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST. • IGNORE: No existence check is performed. • EXPECT_EXIST: The row is expected to exist. If the row exists, the condition is met. If the row does not exist, the condition is not met. • EXPECT_NOT_EXIST: Expect that the row does not exist. If the row does not exist, the condition is met. If the row exists, the condition is not met.
ColumnName	The name of the column.
ColumnValue	The comparison value of the column.
CompareOperator	The relational operator that is used to compare column values. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=).

Parameter	Description
LogicOperator	The logical operator that is used to combine multiple conditions. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. Different logical operators require different number of subconditions. • When the logical operator is NOT, only one subcondition can be added. • When the logical operator is AND or OR, you must add at least two subconditions.
PassIfMissing	Specifies whether to pass the conditional check when a column does not exist in a row. The type of this parameter value is Boolean. The default value is true, which indicates that if the column does not exist in a row, the conditional check is passed, and the row meets the update conditions. When PassIfMissing is set to false, if the column does not exist in a row, the conditional check fails, and the row does not meet the update conditions.
LatestVersionsOnly	Specifies whether to use only the value of the latest version when the column has multiple versions of values. The type of this parameter value is Boolean. The default value is true, which indicates that if the column has multiple versions of data, only the value of the latest version is used for comparison. When LatestVersionsOnly is set to false, the column values of all versions are used for comparison if the column has multiple versions of data. In this case, the conditional check is passed when at least one version of value meets the condition, and the row meets the update conditions.

Examples

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

- Construct a SinglleColumnValueCondition.

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

- Construct a CompositeColumnValueCondition.

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

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

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

2.4.11. Configure local transaction

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

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

You can use local transaction to specify that the operations on a partition key are atomic, which indicates that operations based on the specified partition key either succeed or fail. The isolation level of the local transaction is read committed.

Usage notes

1. Use StartLocalTransaction to create a local transaction based on the specified partition key value and obtain the ID of the local transaction.
2. Read and write data within the local transaction.

   You can call the following operations on the local transaction: GetRow, PutRow, DeleteRow, UpdateRow, BatchWriteRow, and GetRange.
3. Use CommitTransaction to commit local transactions or use AbortTransaction to abort local transactions.

Limits

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

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

    If a transaction is not read or written for more than 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.
  - Up to 4 MB of data can be written to each transaction. The volume of data written to each transaction is calculated in the same way as a regular write request.

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

Parameters

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

Examples

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

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

2. Read and write data within a local transaction.

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

   - Write a row to the local transaction.

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

- Read the row from the local transaction.

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

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

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

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

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

2.4.12. Configure atomic counter

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

Limits

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

overwrite and delete operations on the attribute column A.

- You can perform multiple update operations on the same row by using a BatchWriteRow request. However, if you perform an atomic counter operation on a row, you can perform only one update operation on the row in a BatchWriteRow request.
- Only the value of the latest version of an atomic counter is increased. You cannot increase the value of a specified version of an atomic counter. After you update a row, a new version of data is inserted to the atomic counter in the row.

API operations

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

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

Parameters

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

Examples

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

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

2.4.13. Configure a filter

Tablestore filters query results on the server side before returning the results. Only rows of data that match the filter conditions are returned.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A table is created. Data is written to the table.

Usage notes

When you call the GetRow, BatchGetRow, or GetRange operation to query data, you can use a filter to return only the rows that meet the filter conditions.

The available filters are SingleColumnValueFilter, SingleColumnValueRegexFilter, and CompositeColumnValueFilter.

- SingleColumnValueFilter determines whether to filter a row based only on the values of a reference column.
- SingleColumnValueRegexFilter uses regular expressions to match column values of the String type and extract matching substrings. Then, this filter converts the data type of the extracted substrings to String, Integer, or Double and filters the values after conversion.
- CompositeColumnValueFilter determines whether to filter a row by combining the filter conditions of values of multiple reference columns.

> **Note** For more information about filters, see Configure a filter in Function Introduction.

Limits

- Filters support relational operator-based operations (=, !=, >, >=, <, and <=)and logical operations (NOT, AND, and OR). You can use a combination of up to 10 filter conditions for a filter.
- The reference columns that are used by a filter must be included in the read data. If the specified columns from which data is read do not include reference columns, the filter cannot query the values of reference columns.
- You can use filters by calling the GetRow, BatchGetRow, and GetRange operations, which does not change the native semantics or limits of these operations.

  When you use the GetRange operation, up to 5,000 rows or 4 MB of data can be scanned at a time.

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

Parameters

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

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

Examples

- Construct SingleColumnValueFilter.

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

- Construct SingleColumnValueRegexFilter.

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

- Construct CompositeColumnValueFilter.

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

2.5. Single-row operations

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

> **Note** In this example, pkValue represents the primary key column value. Specify pkValue as required.

PutRow

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

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** ◦ The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table. ◦ For an auto-increment primary key column, you need only to set the value of the auto-increment primary key column to a placeholder. For more information, see Configure an auto-increment primary key column.

Parameter	Description
condition	You can use conditional update to set row existence conditions or column-based conditions for the row. For more information, see Configure conditional update. **Note** ◦ RowExistenceExpectation.IGNORE indicates that new data is inserted into a row regardless of whether the specified row exists or not. If the specified row exists, the existing data is overwritten. ◦ RowExistenceExpectation.EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten. ◦ RowExistenceExpectation.EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist.
column	The attribute column of the row. ◦ An attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType, which is optional), and timestamp (optional). ◦ You can set ColumnType to ColumnType.INTEGER, ColumnType.STRING, ColumnType.BINARY, ColumnType.BOOLEAN, or ColumnType.DOUBLE, which separately indicates INTEGER, STRING (a UTF-8 encoded string), BINARY, BOOLEAN, or DOUBLE. If you want to set the column value type to BINARY, you must set ColumnType to ColumnType.BINARY. If you want to use other types of column values, the setting of ColumnType is optional. ◦ A timestamp is the data version number. For more information, see Data versions and TTL. You can customize a data version number or use the data version number generated by Tablestore. If you do not specify this parameter, the data version number generated by Tablestore is used. ■ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ■ If you specify the version number, make sure that the version number is a 64-bit timestamp accurate to the millisecond within the valid version range.

- Example 1

  The following code provides an example on how to write a row that contains 10 attribute columns and write one version of data for each column. The version number (timestamp) is generated by Tablestore.

```
private static void putRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    // Add attribute columns.
    for (int i = 0; i < 10; i++) {
        rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

- Example 2

  The following code provides an example on how to write a row that contains 10 attribute columns and write three versions of data for each column. In this example, a custom version number (timestamp) is used.

```
private static void putRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    // 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 3

  The following code provides an example on how to write a row that contains 10 attribute columns and write three versions of data for each column when the specified row does not exist. In this example, a custom version number (timestamp) is used.

```
private static void putRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    // Set a conditional update that expects the specified row not to 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

  The following code provides an example on how to write a row that contains 10 attribute columns and write three versions of data for each column when the specified row exists and the value of Col0 is greater than 100. In this example, a custom version number (timestamp) is used.

```
private static void putRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    // Set a conditional update that expects the specified row to exist and the Col0 valu
e of the row to be 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 PutRowRequest(rowPutChange));
}
```

GetRow

You can call this operation to read a row of data.

The following results for the read request may be returned:

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

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
columnsToGet	The names of the columns to read. You can specify the names of primary key columns and the names of attribute columns. If you do not specify this parameter, all data in the row is returned. **Note** ◦ By default, Tablestore returns the data from all columns of the row when you query a row. You can use the columnsToGet parameter to read data from specified columns. For example, if two columns col0 and col1 are included in columnsToGet, the system returns only the values of the two columns. ◦ If you use columnsToGet and filter together, the system first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.

Parameter	Description
maxVersions	The maximum number of versions to read. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If only maxVersions is specified, data of the specified number of versions is returned starting from the latest version. ◦ If only timeRange is specified, all data whose versions are within the specified time range or data of the specified version is returned. ◦ If both maxVersions and timeRange are specified, data of the specified number of versions within the time range is returned starting from the latest version.
timeRange	The range of versions to read or a version of data to read. For more information, see TimeRange. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If only maxVersions is specified, data of the specified number of versions is returned starting from the latest version. ◦ If only timeRange is specified, all data whose versions are within the specified time range or data of the specified version is returned. ◦ If both maxVersions and timeRange are specified, data of the specified number of versions within the time range is returned starting from the latest version. ◦ To query data within a range, you need to set start and end. start indicates the start timestamp. end indicates the end timestamp. The specified range includes the start value and excludes the end value. ◦ To query data of a specific version number, you need to set timestamp. timestamp indicates a specified timestamp. You need only to set one of timestamp and [start, end). The timestamp value ranges from 0 to Long.MAX_VALUE. Unit: milliseconds.

Parameter	Description
filter	The filter used to filter the queried results on the server side. Only rows that meet the specified filter conditions are returned. For more information, see Configure a filter. **Note** If you use columnsToGet and filter together, the system first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.

- Example 1

  The following code provides an example on how to read data of the latest version and the specified columns from a row:

```
private static void getRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the table name and primary key to read a row of data.
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey);
    // Set MaxVersions to 1 to read the latest version of data.
    criteria.setMaxVersions(1);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("Read complete. Result:");
    System.out.println(row);
    // Specify the columns to read.
    criteria.addColumnsToGet("Col0");
    getRowResponse = client.getRow(new GetRowRequest(criteria));
    row = getRowResponse.getRow();
    System.out.println("Read complete. Result:");
    System.out.println(row);
}
```

- Example 2

  The following code provides an example on how to read a row of data while a filter is used.

```
private static void getRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the table name and primary key to read a row of data.
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey);
    // Set MaxVersions to 1 to read the latest version of data.
    criteria.setMaxVersions(1);
    // Set a filter to return the row when the value in Col0 is 0.
    SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
            SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
    // If the Col0 column does not exist, the row is not returned.
    singleColumnValueFilter.setPassIfMissing(false);
    criteria.setFilter(singleColumnValueFilter);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("Read complete. Result:"); ");
    System.out.println(row);
}
```

- Example 3

  The following code provides an example on how to read the data of the Col1 column in a row while a regular expression is used to filter data in the column.

```
private static void getRow(SyncClient client, String pkValue) {
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(tableName);
    // Construct the primary key.
    PrimaryKey primaryKey = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pkValue))
        .build();
    criteria.setPrimaryKey(primaryKey);
    // Set MaxVersions to 1 to read the latest version of data.
    criteria.setMaxVersions(1);
    // Set the filter. Data in the row is returned when cast<int>(regex(Col1) is greater
than 100.
    RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
    SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
        regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromL
ong(100));
    criteria.setFilter(filter);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("Read complete. Result:"); ");
    System.out.println(row);
}
```

UpdateRow

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

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

- Parameters

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

- Example 1

The following code provides an example on how to update columns, delete the data of the specified version from a column, and delete a specified column:

```
private static void updateRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
    // Update columns.
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    // Delete a specified version of a column.
    rowUpdateChange.deleteColumn("Col10", 1465373223000L);
    // Delete a column.
    rowUpdateChange.deleteColumns("Col11");
    client.updateRow(new UpdateRowRequest(rowUpdateChange));
}
```

- Example 2

  The following code provides an example on how to set update conditions:

```
private static void updateRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
    // Set a conditional update that expects the specified row to exist and the Col0 valu
e of the row to be greater than 100.
    Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
    condition.setColumnCondition(new SingleColumnValueCondition("Col0",
            SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(100)));
    rowUpdateChange.setCondition(condition);
    // Update columns.
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    // Delete a specified version of a column.
    rowUpdateChange.deleteColumn("Col10", 1465373223000L);
    // Delete a column.
    rowUpdateChange.deleteColumns("Col11");
    client.updateRow(new UpdateRowRequest(rowUpdateChange));
}
```

DeleteRow

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

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
condition	You can use conditional update to set row existence conditions or columns-based conditions for the row. For more information, see Configure conditional update.

- Example 1

  The following code provides an example on how to delete a row from a table:

```
private static void deleteRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey);
    client.deleteRow(new DeleteRowRequest(rowDeleteChange));
}
```

- Example 2

  The following code provides an example on how to set deletion conditions:

```
private static void deleteRow(SyncClient client, String pkValue) {
    // Construct the primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    // Specify the name of the table.
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey);
    // Set a conditional update that expects a specified row to exist and the Col0 value
of the row to be 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));
}
```

2.6. Multi-row operations

Tablestore SDK for Java provides multi-row operations such as BatchWriteRow, BatchGetRow, GetRange, and createRangeIterator.

BatchWriteRow

You can call this operation to write several rows of data to one or more tables by sending one request. The BatchWriteRow operation consists of PutRow, UpdateRow, and DeleteRow operations. The process of constructing a suboperation is the same as that of calling the PutRow, UpdateRow, and DeleteRow operations. BatchWriteRow supports conditional updates.

> **Note** For more information about how to delete data in batches, see How do I delete multiple rows of data?.

Each suboperation of BatchWriteRow is independently performed, and the result for each suboperation is independently returned.

When multiple rows are written, some rows may fail to be written. If this happens, Tablestore does not return exceptions, but information about the index and error messages of the failed rows is stored in BatchWriteRowResponse. Therefore, when you call the BatchWriteRow operation, you must check the return values. You can use the isAllSucceed method of BatchWriteRowResponse to check whether all operations are successful. If you do not check the return values, some operation failures may be ignored.

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

- Parameters

  For information about parameters, see Single-row operations.

- Examples

  The following code provides an example on how to send a BatchWriteRow request, which includes two PutRow operations, one UpdateRow operation, and one 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 columns.
    for (int i = 0; i < 10; i++) {
        rowPutChange1.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    // Add rowPutChange1 to the 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 columns.
    for (int i = 0; i < 10; i++) {
        rowPutChange2.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    // Add rowPutChange2 to the batch operation.
    batchWriteRowRequest.addRowChange(rowPutChange2);
    // Construct rowUpdateChange.
    PrimaryKeyBuilder pk3Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk3Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk3"));
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, pk3Builder.build())
;
    // Add columns.
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    // Delete a column.
    rowUpdateChange.deleteColumns("Col10");
    // Add rowUpdateChange to the batch operation.
    batchWriteRowRequest.addRowChange(rowUpdateChange);
    // Construct rowDeleteChange.
    PrimaryKeyBuilder pk4Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk4Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk4"));
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, pk4Builder.build())
;
    // Add rowDeleteChange to the batch operation.
    batchWriteRowRequest.addRowChange(rowDeleteChange);
    BatchWriteRowResponse response = client.batchWriteRow(batchWriteRowRequest);
    System.out.println("Whether all operations are successful:" + response.isAllSucceed()
);
    if (!response.isAllSucceed()) {
        for (BatchWriteRowResponse.RowResult rowResult : response.getFailedRows()) {
            System.out.println("Failed rows:" + batchWriteRowRequest.getRowChange(rowResu
lt.getTableName(), rowResult.getIndex()).getPrimaryKey());
            System.out.println("Cause of failures:" + rowResult.getError());
        }
        /**
         * You can use the createRequestForRetry method to construct another request to r
etry the operations on failed rows. Only the retry request is constructed here.
         * We recommend that you use the custom retry policy in Tablestore SDKs as the re
```

```
            * We recommend that you use the custom retry policy in Tablestore SDKs as the re
try method. This feature allows you to retry failed rows after batch operations. After yo
u set the retry policy, you do not need to add retry code to call the operation.
             */
            BatchWriteRowRequest retryRequest = batchWriteRowRequest.createRequestForRetry(re
sponse.getFailedRows());
        }
}
```

For the detailed sample code, visit BatchWriteRow@GitHub.

BatchGetRow

You can call this operation to read several rows of data from one or more tables by sending one request. The BatchGetRow operation consists of multiple GetRow operations. The process of constructing a suboperation is the same as that of calling the GetRow operation.

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

Each suboperation of BatchGetRow is independently performed, and the result for each suboperation is independently returned.

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

- Parameters

  For information about parameters, see Single-row operations.

- Examples

  The following code provides an example on how to configure the version conditions, columns to read, and filters to read 10 rows of data:

```
private static void batchGetRow(SyncClient client) {
    MultiRowQueryCriteria multiRowQueryCriteria = new MultiRowQueryCriteria(TABLE_NAME);
    // Add 10 rows.
    for (int i = 0; i < 10; i++) {
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder()
;
        primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromStrin
g("pk" + i));
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        multiRowQueryCriteria.addRow(primaryKey);
    }
    // Add conditions.
    multiRowQueryCriteria.setMaxVersions(1);
    multiRowQueryCriteria.addColumnsToGet("Col0");
    multiRowQueryCriteria.addColumnsToGet("Col1");
    SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
            SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
    singleColumnValueFilter.setPassIfMissing(false);
    multiRowQueryCriteria.setFilter(singleColumnValueFilter);
    BatchGetRowRequest batchGetRowRequest = new BatchGetRowRequest();
    // BatchGetRow allows you to read data from multiple tables. A single multiRowQueryCr
iteria specifies a query condition for one table. You can add multiple multiRowQueryCrite
ria conditions to read data from multiple tables.
    batchGetRowRequest.addMultiRowQueryCriteria(multiRowQueryCriteria);
    BatchGetRowResponse batchGetRowResponse = client.batchGetRow(batchGetRowRequest);
    System.out.println("Whether all operations are successful:" + batchGetRowResponse.isA
llSucceed());
    if (!batchGetRowResponse.isAllSucceed()) {
        for (BatchGetRowResponse.RowResult rowResult : batchGetRowResponse.getFailedRows(
)) {
            System.out.println("Failed rows:" + batchGetRowRequest.getPrimaryKey(rowResul
t.getTableName(), rowResult.getIndex()));
            System.out.println("Cause of failures:" + rowResult.getError());
        }
        /**
         * You can use the createRequestForRetry method to construct another request to r
etry the operations on failed rows. Only the retry request is constructed here.
         * We recommend that you use the custom retry policy in Tablestore SDKs as the re
try method. This feature allows you to retry failed rows after batch operations. After yo
u set the retry policy, you do not need to add retry code to call the operation.
         */
        BatchGetRowRequest retryRequest = batchGetRowRequest.createRequestForRetry(batchG
etRowResponse.getFailedRows());
    }
}
```

For the detailed sample code, visit BatchGetRow@GitHub.

GetRange

You can call this operation to read data in a forward and backward direction within a specified range.

You can also limit the number of rows that you want to read. If the range is large and the number of scanned rows or the volume of data exceeds the upper limit, the scan stops, and the read rows and the next primary key information are returned. 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 information returned by the previous operation.

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

The GetRange operation may stop and return data in the following situations:

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

Parameter	Description
tableName	The name of the table.
direction	The direction for reading. ◦ If this parameter is set to FORWARD, the value of the start primary key must be less than that of the end primary key. The returned rows are sorted in ascending order based on their primary key values. ◦ If the value is BACKWARD, the value of the start primary key must be greater than that of the end primary key. The returned rows are sorted in descending order based on their primary key values. For example, if you set direction to FORWARD for a table that has two primary keys A and B and the value of A is less than that of B, the rows whose primary key values are equal to or greater than the value of A but less than the value of B are returned in ascending order from A to B. If you set direction to BACKWARD, the rows whose primary key values are equal to or less than the value of B and greater than the value of A are returned in descending order from B to A.

Parameter	Description
inclusiveStartPrimaryKey	The start primary key and end primary key of the range to be read. The start and end primary keys must be valid primary keys or virtual points composed of the INF_MIN and INF_MAX type data. The number of columns for each virtual point must be the same as that of each primary key. INF_MIN indicates an infinitely small value. All other values of other types are greater than INF_MIN. INF_MAX indicates an infinitely great value. All other values of other types are less than INF_MAX. ○ inclusiveStartPrimaryKey indicates the start primary key. If a row that contains the start primary key exists, the row is included in the response. ○ exclusiveEndPrimaryKey indicates the end primary key. The row that corresponds to the end primary key is not contained in the response regardless of whether the row exists. The rows in the data table are sorted in ascending order based on the primary key values. The range to read is a left-closed and right-open interval. If data is read in the forward direction, the rows whose primary keys are greater than or equal to the start primary key but less than the end primary key are returned.
exclusiveEndPrimaryKey	
limit	The maximum number of rows that can be returned. The value of this parameter must be greater than 0. An operation stops after the maximum number of rows returned in the forward or backward direction is reached, even if some rows within the specified range are not returned. You can use the value of nextStartPrimaryKey returned in the response to read data in the next request.

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

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

- Example 1

The following code provides an example on how to read data within a specified range in the forward direction. If the nextStartPrimaryKey value is null in the response, all data in the specified range is read.

```
private static void getRange(SyncClient client, String startPkValue, String endPkValue) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    // Specify the start primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(st
artPkValue));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    // Specify the end primary key.
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(en
dPkValue));
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("GetRange result:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the NextStartPrimaryKey value is not null, continue the read operation.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- Example 2

  The following code provides an example on how to read data in the forward direction within the range determined by the value of the first primary key column. The value of the second primary key column in the start primary key is INF_MIN. The value of the second primary key column in the end primary key is INF_MAX. If the nextStartPrimaryKey value is null in the response, all data in the specified range is read.

```
private static void getRange(SyncClient client, String startPkValue, String endPkValue) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    // Specify the start primary key to. In this example, two primary keys are used.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1, PrimaryKeyValue.fromString(s
tartPkValue)); // Set the value of the first primary key column to a specific value.
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2, PrimaryKeyValue.INF_MIN); //
Set the value of the second primary key column to an infinitely small value.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    // Specify the end primary key.
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1, PrimaryKeyValue.fromString(e
ndPkValue)); // Set the value of the first primary key column to a specific value.
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2, PrimaryKeyValue.INF_MAX); //
Set the value of the second primary key column to an infinitely great value.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("GetRange result:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- Example 3

  The following code is an example on how to read the data in the Col1 column within the primary key range [pk:2020-01-01.log to pk:2021-01-01.log) and filter the data in that column.

```
private static void getRange(SyncClient client) {
    RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(TABLE_NAME);
    // Set the primary key range to [pk:2020-01-01.log, pk:2021-01-01.log), and the read
range is in a left-closed and right-open interval.
    PrimaryKey pk0 = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("pk", PrimaryKeyValue.fromString("2020-01-01.log"))
        .build();
    PrimaryKey pk1 = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("pk", PrimaryKeyValue.fromString("2021-01-01.log"))
        .build();
    criteria.setInclusiveStartPrimaryKey(pk0);
    criteria.setExclusiveEndPrimaryKey(pk1);
    // Set the latest version to be read.
    criteria.setMaxVersions(1);
    // Set the filter. Data in the row is returned when cast<int>(regex(Col1) is greater
than 100.
    RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
    SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
        regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN,ColumnValue.fromLo
ng(100));
    criteria.setFilter(filter);
    while (true) {
        GetRangeResponse resp = client.getRange(new GetRangeRequest(criteria));
        for (Row row : resp.getRows()) {
            // do something
            System.out.println(row);
        }
        if (resp.getNextStartPrimaryKey() != null) {
            criteria.setInclusiveStartPrimaryKey(resp.getNextStartPrimaryKey())
        }
    }
}
```

For the detailed sample code, visit GetRange@GitHub.

createRangeIterator

You can call this operation to read data by iteration.

```
private static void getRangeByIterator(SyncClient client, String startPkValue, String endPk
Value) {
    RangeIteratorParameter rangeIteratorParameter = new RangeIteratorParameter(TABLE_NAME);
    // Specify the start primary key.
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(star
tPkValue));
    rangeIteratorParameter.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    // Specify the end primary key.
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(endP
kValue));
    rangeIteratorParameter.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeIteratorParameter.setMaxVersions(1);
    Iterator<Row> iterator = client.createRangeIterator(rangeIteratorParameter);
    System.out.println("Results obtained when Iterator is used to implement GetRange:");
    while (iterator.hasNext()) {
        Row row = iterator.next();
        System.out.println(row);
    }
}
```

For the detailed sample code, visit GetRangeByIterator@GitHub.

2.7. Incremental data operations

Tablestore provides the ListStream and DescribeStream operations for streams, and GetShardIterator and GetShardRecord operations for shards.

ListStream

You can call the ListStream operation to list all streams enabled for a table on an instance.

Example

List the information about all streams enabled for a table.

```
private static void listStream(SyncClient client, String tableName) {
    ListStreamRequest listStreamRequest = new ListStreamRequest(tableName);
    ListStreamResponse result = client.listStream(listStreamRequest);
}
```

DescribeStream

You can call the DescribeStream operation to query the creationTime, expirationTime, status, shards, and ID of the next start shard (if shards are not returned) of a stream.

Example 1

Obtain the information about all shards of a stream.

```
    private static void describeStream(SyncClient client, String streamId) {
        DescribeStreamRequest desRequest = new DescribeStreamRequest(streamId);
        DescribeStreamResponse desStream = client.describeStream(desRequest);
    }
```

Example 2

Set the ID of the start shard (InclusiveStartShardId) and the maximum number of shards to return each time.

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

GetShardIterator

You can call the GetShardIterator operation 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(getShardIteratorReques
t);
    }
```

GetStreamRecord

You can call the GetStreamRecord operation to obtain the update records 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(shardIterat
or);
        streamRecordRequest.setLimit(100);
        GetStreamRecordResponse streamRecordResponse = client.getStreamRecord(streamRecordR
equest);
        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.getNextShardIterator());
    }
```

2.8. Search Index

2.8.1. Create search indexes

You can call the CreateSearchIndex operation to create one or more search indexes for a data table.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created for which the value of the timeToLive parameter is set to -1 and the value of the maxVersions parameter is set to 1.

Parameters

When you create a search index, you must configure the tableName, indexName, and indexSchema parameters. You must also configure the fieldSchemas, indexSetting, indexSort, and timeToLive parameters in indexSchema. The following table describes the parameters.

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

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

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

2.8.2. List search indexes

You can call the ListSearchIndex operation to obtain indexes created in the current instance or associated with a table.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

2.8.3. 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.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

2.8.4. Delete search indexes

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

2.8.5. Term query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

2.8.6. Terms query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
query	The type of the query. Set the value to TermsQuery.
fieldName	The name of the field that you want to match.

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

Examples

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

2.8.7. Match all query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
query	The query type. Set the query type to MatchAllQuery.
tableName	The name of the data table.

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

Examples

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

2.8.8. Match query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.

- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
fieldName	The name of the column that you want to query. Match query applies to TEXT columns.
text	The keyword that is used to match the column values when you perform a match query. If the column to query is a TEXT column, the keyword is tokenized into multiple tokens based on the analyzer that you specify when you create the search index. By default, single-word tokenization is performed if you do not specify the analyzer when you create the search index. For example, if you set the tokenization method to single-word tokenization and use "this is" as a search keyword, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
query	The query type, which is set to matchQuery.
offset	The position from which the current query starts.
limit	The maximum number of rows that you want the current query to return. To query only the number of matched rows without returning specific data, you can set limit to 0. This way, Tablestore returns the number of matched rows instead of specific data from the table.
minimumShouldMatch	The minimum number of matched tokens contained in a column value. A row is returned only when the value of the fieldName column in the row contains at least the minimum number of matched tokens. **Note** minimumShouldMatch must be used together with the OR logical operator.
operator	The logical operator. By default, OR is used as the logical operator, which specifies that a row matches the query conditions when one of the tokens is matched. If you set the operator to AND, the row meets the query conditions only when the column value contains all tokens.

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

Examples

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

2.8.9. Phrase query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

2.8.10. Prefix query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
query	The type of the query. Set the query type to PrefixQuery.

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

Examples

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

2.8.11. Range query

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

2.8.12. Wildcard query

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

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

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

2.8.13. Geo query

Geo query includes the following query types: geo-distance query (GeoDistanceQuery), geo-bounding box query (GeoBoundingBoxQuery), and geo-polygon query (GeoPolygonQuery).

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Geo-distance query

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

- Parameter

Parameter	Description
fieldName	The name of the column. Set the query type to Geopoint.
centerPoint	The coordinate pair of the central point. The coordinate pair consists of latitude and longitude values. This parameter value must be in the format of `latitude,longitude`. Valid values of the latitude: [-90,+90]. Valid values of the longitude: [-180,+180]. Example: `35.8,-45.91`.
distanceInMeter	The radius of the circle centered on the specified location. Type: DOUBLE. Unit: meter.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
query	The query statement for the search index. Set the query type to GeoDistanceQuery.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

- Examples

  The following code provides an example on how to search the table for rows where the value of Col_GeoPoint falls within a specified distance from a specified central point:

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

Geo-bounding box query

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

- Parameters
- Examples

  The data type of Col_GeoPoint is GeoPoint. The coordinate pair of the upper-left corner is "10,0". The coordinate pair of the lower-right corner is "0,10". The following code provides an example on how to obtain the rows where the value of Col_GeoPoint falls within the rectangular geographic area:

Geo-polygon query

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

- Parameters
- Examples

The following code provides an example on how to query the table for rows where the value of Col_GeoPoint falls within a specified geographic polygon area:

2.8.14. Boolean query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
mustQueries	The list of subqueries that the query results must match. This parameter is equivalent to the AND operator.
mustNotQueries	The list of subqueries that the query results must not match. This parameter is equivalent to the NOT operator.
filterQueries	The list of subqueries. Only rows that match all subfilters are returned. filter is similar to query except that filter does not calculate the relevance score based on the number of subfilters that the row matches.
shouldQueries	The list of subqueries that the query results can or cannot match. This parameter is equivalent to the OR operator. Only rows that meet the minimum number of subquery conditions specified by shouldQueries are returned. A higher overall relevance score indicates that more subquery conditions specified by shouldQueries are met.
minimumShouldMatch	The minimum number of subquery conditions specified by shouldQueries that the rows must meet. If no other subquery conditions except the subquery conditions that are specified by shouldQueries are specified, the default value of the minimumShouldMatch parameter is 1. If other subquery conditions, such as subquery conditions specified by mustQueries, mustNotQueries, and filterQueries are specified, the default value of the minimumShouldMatch parameter is 0.

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

Examples

- Example 1:

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

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

- Example 2

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

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

- Example 3

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

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

- Example 4

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

```
/**
 * (col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6))
 * In the preceding example, each subquery of the BoolQuery type is connected by AND or O
R.
 * @param client
*/
private static void boolQuery2(SyncClient client){
        // Condition 1: col2 < 4
```

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

```
        System.out.println(response.getRows());
    }
```

2.8.15. Nested query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
path	The path of the nested field. The path is similar to the tree structure. For example, news.title indicates the title subcolumn in the nested field named news.
query	The query implemented on the subcolumn in the nested field. The query can be of any query type.
scoreMode	Specifies which value is used to calculate the score when the field contains multiple values.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
tableName	The name of the table.
indexName	The name of the search index.
columnsToGet	Specifies whether to return all columns of each matched row. You can configure returnAll and columns for this parameter. By default, returnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If returnAll is set to true, all columns are returned.

Examples

The following examples show how to use nested queries.

- Example 1

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

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

```
lumns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Set the setColumn
s parameter to return specified columns.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total num
ber of matched columns but not returned columns.
    System.out.println("Row: " + resp.getRows());
}
```

- Example 2

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

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

2.8.16. Sorting and pagination

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

Index presorting

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

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

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

Specify a sorting method

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

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

- ScoreSort

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

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

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

- PrimaryKeySort

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

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

- FieldSort

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

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

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

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

- GeoDistanceSort

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

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

Specify a pagination method

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

- Use the limit and offset parameters

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

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

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

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

- Use a token

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

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

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

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

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

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

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

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

2.8.17. Exists query

Exists query is also called NULL query or NULL-value query. This query is used in sparse data to determine whether a column of a row exists. For example, you can query the rows in which the value of the address column is not empty.

> **Note**
>
> - If you want to check whether a column contains empty values, you must use ExistsQuery together with mustNotQueries of BoolQuery.
> - If one of the following conditions is met, the system considers that a column does not to exist. In this example, the city column is used.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the city column does not exist in the data table, the search index considers that the city column does not exist.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the value of the city column is an empty array in the data table ("city" = "[]"), the search index considers that the city column does not exist.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
fieldName	The name of the column.
query	The query type. Set the value to ExistsQuery.
getTotalCount	Specifies whether to return the total number of rows that meet the query conditions. The default value of this parameter is false, which indicates that the total number of rows that meet the query conditions is not returned. If you set this parameter to true, the query performance is compromised.
tableName	The name of the data table.
indexName	The name of the search index.

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

Examples

```
/**
 * Use ExistsQuery to query the rows in which the address column is not empty.
 * @param client
 */
private static void existQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    ExistsQuery existQuery = new ExistsQuery(); // Set the query type to ExistsQuery.
    existQuery.setFieldName("address");
    searchQuery.setQuery(existQuery);
    //searchQuery.setGetTotalCount(true); // Specify that the total number of rows that mee
t the query conditions is returned.
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    // You can use the columnsToGet parameter to specify the columns that you want to retur
n or specify that all columns are returned. If you do not specify this parameter, only the
primary key columns are returned.
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns.
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); // Specify the columns
that you want to return.
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); // Specify that the total
number of rows that meet the query conditions instead of the number of returned rows is dis
played.
    System.out.println("Row: " + resp.getRows());
}
```

2.8.18. Collapse (distinct)

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

2.8.19. Aggregation

You can perform aggregation operations to obtain the minimum value, maximum value, sum, average, count and distinct count of rows, percentile statistics, and rows in each group. You can also perform aggregation operations to group results by field value, range, geographical location, filter, or histogram, and perform nested queries. You can perform multiple aggregation operations for complex queries.

Minimum value

The aggregation method that can be used to return the minimum value of a field. This method can be used in a similar manner as the SQL MIN function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 *  The price of each product is listed in the product table. Query the minimum price of
the products that are produced in Zhejiang.
 *  SQL statement: SELECT min(column_price) FROM product where place_of_production = "Zhe
jiang".
 */
public void min(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production","Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead o
f specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.min("min_agg_1", "column_price").miss
ing(100))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("min_agg_1"
).getValue());
}
```

Maximum value

The aggregation method that can be used to return the maximum value of a field. This method can be used in a similar manner as the SQL MAX function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The price of each product is listed in the product table. Query the maximum price of t
he products that are produced in Zhejiang.
 * SQL statement: SELECT max(column_price) FROM product where place_of_production = "Zhej
iang".
 */
public void max(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead o
f specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.max("max_agg_1", "column_price").miss
ing(0))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsMaxAggregationResult("max_agg_1"
).getValue());
}
```

Sum

The aggregation method that can be used to return the sum of all values for a numeric field. This method can be used in a similar manner as the SQL SUM function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The sales volume of each product is listed in the product table. Query the total numbe
r of the sold products that are produced in Zhejiang. The value of the missing parameter
is set to 10.
 * SQL statement: SELECT sum(column_price) FROM product where place_of_production = "Zhej
iang".
 */
public void sum(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead o
f specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.sum("sum_agg_1", "column_number").mis
sing(10))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("sum_agg_1"
).getValue());
}
```

Average

The aggregation method that can be used to return the average of all values for a numeric field. This method is used in a similar manner as the SQL AVG function.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The sales volume of each product is listed in the product table. Query the average pri
ce of the products that are produced in Zhejiang.
 * SQL statement: SELECT avg(column_price) FROM product where place_of_production = "Zhej
iang".
 */
public void avg(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "Zhejiang"))
                .limit(0) // If you want to obtain only the aggregation results instead o
f specific data, you can set limit to 0 to improve query performance.
                .addAggregation(AggregationBuilders.avg("avg_agg_1", "column_number"))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsAvgAggregationResult("avg_agg_1"
).getValue());
}
```

Count

The aggregation method that can be used to return the total number of values for a specified field or the total number of rows in a table. This method can be used in a similar manner as the SQL COUNT function.

> **Note** You can use one of the following methods to query the total number of rows in a table or the total number of rows that match the query conditions:
>
> - Use the count feature of aggregation. Set the count parameter to * in the request.
> - Use the query feature to obtain the number of rows that match the query conditions. Set the setGetTotalCount parameter to true in the query. Use MatchAllQuery to obtain the total number of rows in a table.
>
> You can use the name of a column as the value of the count expression to query the number of rows that contain the column in a table. This method is suitable for scenarios that involve sparse columns.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.

Parameter	Description
fieldName	The name of the field that is used for the aggregation operation. Only the following data types are supported: LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT.

- Examples

```
/**
 * Punishment records of merchants are recorded in the merchant table. You can query the number of merchants who are located in Zhejiang and for whom punishment records exist. If no punishment records exist for a merchant, the field that corresponds to punishment records also does not exist for the merchant.
 * SQL statement: SELECT count(column_history) FROM product where place_of_production = "Zhejiang".
 */
public void count(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place", "Zhejiang"))
                .limit(0)
                .addAggregation(AggregationBuilders.count("count_agg_1", "column_history"))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsCountAggregationResult("count_agg_1").getValue());
}
```

Distinct count

The aggregation method that can be used to return the number of distinct values for a field. This method can be used in a similar manner as the SQL COUNT(DISTINCT) function.

> **Note** The number of distinct values is an approximate number.
>
> - If the total number of rows before the distinct count feature is used is less than 10,000, the calculated result is an exact value.
> - If the total number of rows before the distinct count feature is used is greater than or equal to 100 million, the error rate is approximately 2%.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the following data types are supported: LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * Query the number of distinct provinces from which the products are produced.
 * SQL statement: SELECT count(distinct column_place) FROM product.
 */
public void distinctCount(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.distinctCount("dis_count_agg_1", "col
umn_place"))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
dis_count_agg_1").getValue());
}
```

Percentile statistics

A percentile value indicates the relative position of a value in a dataset. For example, when you collect statistics for the response time of each request during the routine O&M of your system, you must analyze the response time distribution by using percentiles such as p25, p50, p90, and p99.

> **Note** To improve the accuracy of the results, we recommend that you specify extreme percentile values such as p1 and p99. If you use extreme percentile values instead of other values such as p50, the returned results are more accurate.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the LONG and DOUBLE data types are supported.
percentiles	The percentiles such as p50, p90, and p99. You can specify one or more percentiles.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * Analyze the distribution of the response time of each request that is sent to the syst
em by using percentiles.
 */
public  void percentilesAgg(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addAggregation(AggregationBuilders.percentiles("percentilesAgg", "latenc
y")
                    .percentiles(Arrays.asList(25.0d, 50.0d, 99.0d))
                    .missing(1.0))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the analysis results.
    PercentilesAggregationResult percentilesAggregationResult = resp.getAggregationResult
s().getAsPercentilesAggregationResult(
        "percentilesAgg");
    for (PercentilesAggregationItem item : percentilesAggregationResult.getPercentilesAgg
regationItems()) {
        System.out.println("key: " + item.getKey() + " value:" + item.getValue().asDouble
());
    }
}
```

Group by field value

The aggregation method that can be used to group query results based on field values. The values that are the same are grouped together. The identical value of each group and the number of identical values in each group are returned.

> **Note** The calculated number may be different from the actual number if the number of values in a group is very large.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the following data types are supported: LONG, DOUBLE, BOOLEAN, and KEYWORD.
groupBySorter	The sorting rules for items in a group. By default, group items are sorted in descending order. If you configure multiple sorting rules, data is sorted based on the order in which the rules are configured. Supported parameters: ◦ Sort by value in alphabetical order ◦ Sort by value in reverse alphabetical order ◦ Sort by row count in ascending order ◦ Sort by row count in descending order ◦ Sort the values that are obtained from sub-aggregation results in ascending order ◦ Sort the values that are obtained from sub-aggregation results in descending order
size	The number of returned groups. Maximum value: 2000. If the number of groups exceeds 2,000, only the first 2,000 groups are returned.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results. ◦ Scenario Query the number of products in each category, and the maximum and minimum product prices in each category. ◦ Method Group query results by product category to obtain the number of products in each category. Then, perform two sub-aggregation operations to obtain the maximum and minimum product prices in each category. ◦ Examples ■ Fruits: 5. The maximum price is 15. The minimum price is 3. ■ Toiletries: 10. The maximum price is 98. The minimum price is 1. ■ Electronic devices: 3. The maximum price is 8,699. The minimum price is 2,300. ■ Other products: 15. The maximum price is 1,000. The minimum price is 80.

- Example 1:

```
/**
 * Query the number of products, and the maximum and minimum product prices in each categ
ory.
 * Example of returned results: Fruits: 5. The maximum price is 15, and the minimum price
is 3. Toiletries: 10. The maximum price is 98, and the minimum price is 1. Electronic dev
ices: 3. The maximum price is 8,699, and the minimum price is 2,300. Other products: 15.
The maximum price is 1,000, and the minimum price is 80.
 */
public void groupByField(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByField("name1", "column_type")
                    .addSubAggregation(AggregationBuilders.min("subName1", "column_price"
))
                    .addSubAggregation(AggregationBuilders.max("subName2", "column_price"
))
                )
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("
name1").getGroupByFieldResultItems()) {
        // Display values.
        System.out.println(item.getKey());
        // Display the number of rows.
        System.out.println(item.getRowCount());
        // Display the minimum prices.
        System.out.println(item.getSubAggregationResults().getAsMinAggregationResult("sub
Name1").getValue());
        // Display the maximum prices.
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("sub
Name2").getValue());
    }
}
```

- Example 2

```
    /**
     * Group results based on multiple fields.
     * Search indexes do not support the GROUP BY clause for multiple fields in SQL. You
can use nested GroupBy parameters to obtain the results that are the same as those obtain
ed by using the GROUP BY clause for multiple fields.
     * SQL statement: select a,d, sum(b),sum(c) from user group by a,d.
     */
```

```
    public void GroupByMultiField() {
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .returnAllColumns(true)   // You can set returnAllColumns to false and specif
y a value for addColumesToGet to have better query performance.
            //.addColumnsToGet("col_1","col_2")
            .searchQuery(SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())   // Specify query conditions. Query con
ditions can be used in the same manner as the WHERE clause in SQL. You can use QueryBuild
ers.bool() to perform nested queries.
                .addGroupBy(
                    GroupByBuilders
                        .groupByField("unique name_1", "field_a")
                        .size(20)
                        .addSubGroupBy(
                            GroupByBuilders
                                .groupByField("unique name_2", "field_d")
                                .size(20)
                                .addSubAggregation(AggregationBuilders.sum("unique name_3
", "field_b"))
                                .addSubAggregation(AggregationBuilders.sum("unique name_4
", "field_c"))
                        )
                )
                .build())
            .build();
        SearchResponse response = client.search(searchRequest);
        // Query rows that meet the specified conditions.
        List<Row> rows = response.getRows();
        // Obtain the aggregation results.
        GroupByFieldResult groupByFieldResult1 = response.getGroupByResults().getAsGroupB
yFieldResult("unique name_1");
        for (GroupByFieldResultItem resultItem : groupByFieldResult1.getGroupByFieldResul
tItems()) {
            System.out.println("field_a key:" + resultItem.getKey() + " Count:" + resultI
tem.getRowCount());
            // Obtain the sub-aggregation results.
            GroupByFieldResult subGroupByResult = resultItem.getSubGroupByResults().getAs
GroupByFieldResult("unique name_2");
            for (GroupByFieldResultItem item : subGroupByResult.getGroupByFieldResultItem
s()) {
                System.out.println("field_a " + resultItem.getKey() + " field_d key:" + i
tem.getKey() + " Count: " + item.getRowCount());
                double sumOf_field_b = item.getSubAggregationResults().getAsSumAggregatio
nResult("unique name_3").getValue();
                double sumOf_field_c = item.getSubAggregationResults().getAsSumAggregatio
nResult("unique name_4").getValue();
                System.out.println("sumOf_field_b:" + sumOf_field_b);
                System.out.println("sumOf_field_c:" + sumOf_field_c);
            }
        }
    }
```

- Example 3

```
    /**
     * Configure sorting rules for aggregation.
     * Method: Configure sorting rules by specifying GroupBySorter. If you configure mult
iple sorting rules, data is sorted based on the order in which the rules are configured.
GroupBySorter supports sorting in ascending or descending order.
     * By default, data is sorted by row count in descending order. You can sort data by
using GroupBySorter.rowCountSortInDesc().
     */
    public void groupByFieldWithSort(SyncClient client) {
        // Create a query statement.
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .searchQuery(
                SearchQuery.newBuilder()
                    .query(QueryBuilders.matchAll())
                    .limit(0)
                    .addGroupBy(GroupByBuilders
                        .groupByField("name1", "column_type")
                        //.addGroupBySorter(GroupBySorter.subAggSortInAsc("subName1")) //
Sort data in ascending order based on the values that are obtained from sub-aggregation r
esults.
                        .addGroupBySorter(GroupBySorter.groupKeySortInAsc())           //
Sort data in ascending order based on the values that are obtained from aggregation resul
ts.
                        //.addGroupBySorter(GroupBySorter.rowCountSortInDesc())        //
Sort data in descending order based on the number of rows that are obtained from the aggr
egation results.
                        .size(20)
                        .addSubAggregation(AggregationBuilders.min("subName1", "column_pr
ice"))
                        .addSubAggregation(AggregationBuilders.max("subName2", "column_pr
ice"))
                    )
                    .build())
            .build();
        // Execute the query statement.
        SearchResponse resp = client.search(searchRequest);
    }
```

Group by range

The aggregation method that can be used to group query results based on the value ranges of a field. Field values that are within a specified range are grouped together. The number of values in each range is returned.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.

Parameter	Description
fieldName	The name of the field that is used for the aggregation operation. Only the LONG and DOUBLE data types are supported.
range[double_from, double_to)	The value ranges for grouping. The value range can start from Double.MIN_VALUE and end with Double.MAX_VALUE.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results. For example, after you group query results by sales volume and by province, you can obtain the province that has the largest proportion of sales volume in a specified range. You must specify a value for GroupByField in GroupByRange to perform this query.

- Examples

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

Group by geographical location

The aggregation method that can be used to group query results based on geographical locations to a central point. Query results in distances that are within a specified range are grouped together. The number of items in each range is returned.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used for the aggregation operation. Only the GEOPOINT data type is supported.
origin(double lat, double lon)	The longitude and latitude of the central point. double lat specifies the latitude of the central point. double lon specifies the longitude of the central point.
range[double_from, double_to)	The distance ranges that are used for grouping. Unit: meters. The start value of the value range can be Double.MIN_VALUE and the end value can be Double.MAX_VALUE.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results.

- Examples

```
/**
 * Group users based on geographical locations to a Wanda Plaza to obtain the number of u
sers in each distance range. The distance ranges are [0, 1000), [1000, 5000), and [5000,
Double.MAX_VALUE). Unit: meters.
 */
public void groupByGeoDistance(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByGeoDistance("name1", "column_geo_point")
                    .origin(3.1, 6.5)
                    .addRange(0, 1000)
                    .addRange(1000, 5000)
                    .addRange(5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results.
    for (GroupByGeoDistanceResultItem item : resp.getGroupByResults().getAsGroupByGeoDist
anceResult("name1").getGroupByGeoDistanceResultItems()) {
       // Display the number of rows.
        System.out.println(item.getRowCount());
    }
}
```

Group by filter

The aggregation method that can be used to filter the query results and group them together to obtain the number of results that match each filter. Results are returned in the order in which the filters are specified.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
filter	The filters that can be used for the query. Results are returned in the order in which the filters are specified.
subAggregation and subGroupBy	The sub-aggregation operation. You can perform the sub-aggregation operation based on the grouping results.

- Examples

```
/**
 * Specify the following filters to obtain the number of items that match each filter: Th
e sales volume exceeds 100, the place of origin is Zhejiang, and the description contains
Hangzhou.
 */
public void groupByFilter(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByFilter("name1")
                    .addFilter(QueryBuilders.range("number").greaterThanOrEqual(100))
                    .addFilter(QueryBuilders.term("place","Zhejiang"))
                    .addFilter(QueryBuilders.match("text","Hangzhou"))
                )
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the aggregation results based on the order of filters.
    for (GroupByFilterResultItem item : resp.getGroupByResults().getAsGroupByFilterResult
("name1").getGroupByFilterResultItems()) {
        // Display the number of rows.
        System.out.println(item.getRowCount());
    }
}
```

Query by histogram

The aggregation method that can be used to group query results based on specific data intervals. Field values that are within the same range are grouped together. The value range of each group and the number of values in each group are returned.

- Parameters

Parameter	Description
groupByName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
fieldName	The name of the field that is used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
interval	The data interval that is used to obtain aggregation results.
fieldRange[min,max]	The range that is used together with the interval parameter to limit the number of groups. The value that is calculated by using the `(fieldRange.max-fieldRange.min)/interval` formula cannot exceed 2,000.

Parameter	Description
minDocCount	The minimum number of rows. If the number of rows in a group is less than the minimum number of rows, the aggregation results for the group are not returned.
missing	The default value for the field that is used to perform the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * Collect statistics on the distribution of users by age group.
 */
public static void groupByHistogram(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addGroupBy(GroupByBuilders
                    .groupByHistogram("groupByHistogram", "age")
                    .interval(10)
                    .minDocCount(0L)
                    .addFieldRange(0, 99))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = ots.search(searchRequest);
    // Obtain the results that are returned when the aggregation operation is performed.
    GroupByHistogramResult results = resp.getGroupByResults().getAsGroupByHistogramResult
("groupByHistogram");
    for (GroupByHistogramItem item : results.getGroupByHistogramItems()) {
        System.out.println("key:" + item.getKey().asLong() + " value:" + item.getValue())
;
    }
}
```

Query the rows that are obtained from the results of an aggregation operation in each group

After you group query results, you can query the rows in each group. This method can be used in a similar manner as ANY_VALUE(field) in MySQL.

> **Note** When you query the rows that are obtained from the results of an aggregation operation in each group, the returned results contain only the primary key information if the search index contains the NESTED, GEOPOINT, or ARRAY field. To obtain the required field, you must query the data table.

- Parameters

Parameter	Description
aggregationName	The unique name of the aggregation operation. You can query the results of a specific aggregation operation based on this name.
limit	The maximum number of rows that can be returned for each group. By default, only one row of data is returned.
sort	The sorting method that is used to sort data in groups.
columnsToGet	The fields that you want to return. Only fields in search indexes are supported. ARRAY, GEOPOINT, and NESTED fields are not supported.

- Examples

```
/**
 * An activity application form of a school contains fields in which information such as
the names of students, classes, head teachers, and class presidents can be specified. You
can group students by class to view the application statistics and the property informati
on of each class.
 * SQL statement: select className, teacher, monitor, COUNT(*) as number from table GROUP
BY className.
 */
public void testTopRows(SyncClient client) {
    SearchRequest searchRequest = SearchRequest.newBuilder()
            .indexName("indexName")
            .tableName("tableName")
            .searchQuery(
                    SearchQuery.newBuilder()
                            .query(QueryBuilders.matchAll())
                            .limit(0)
                            .addGroupBy(GroupByBuilders.groupByField("groupName", "classN
ame")
                                    .size(5)
                                    .addSubAggregation(AggregationBuilders.topRows("topRo
wsName")
                                            .limit(1)
                                            .sort(new Sort(Arrays.asList(new FieldSort("t
eacher", SortOrder.DESC)))) // Sort rows by teacher in descending order.
                                    )
                            )
                            .build())
            .addColumnsToGet(Arrays.asList("teacher", "monitor"))
            .build();
    SearchResponse resp = client.search(searchRequest);
    List<GroupByFieldResultItem> items = resp.getGroupByResults().getAsGroupByFieldResult
("groupName").getGroupByFieldResultItems();
    for (GroupByFieldResultItem item : items) {
        String className = item.getKey();
        long number = item.getRowCount();
        List<Row> topRows = item.getSubAggregationResults().getAsTopRowsAggregationResult
("topRowsName").getRows();
        Row row = topRows.get(0);
        String teacher = row.getLatestColumn("teacher").getValue().asString();
        String monitor = row.getLatestColumn("monitor").getValue().asString();
    }
}
```

Nesting

GroupBy supports nesting. You can perform sub-aggregation operations by using GroupBy.

You can use nesting to perform sub-aggregation operations in a group. For example, you can perform nesting aggregation operations up to two levels.

- GroupBy + SubGroupBy: Items are grouped by province and by city to obtain data for each city in each province.
- GroupBy + SubAggregation: Items are grouped by province to obtain the maximum value of a metric for each province.

Note To ensure high performance of queries and perform GroupBy operations, you can specify only a small number of levels for nesting. For more information, see Search index limits.

Examples

```
/**
 * Perform nesting-based aggregation.
 * Two aggregations and one GroupByField attribute are specified in the outermost level. Tw
o sub-aggregations and one GroupByRange attribute are specified in GroupByField.
 */
public void subGroupBy(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .indexName("index_name")
        .tableName("table_name")
        .returnAllColumns(true)
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.match("textField", "hello"))
                .limit(10)
                .addAggregation(AggregationBuilders.min("name1", "fieldName1"))
                .addAggregation(AggregationBuilders.max("name2", "fieldName2"))
                .addGroupBy(GroupByBuilders
                    .groupByField("name3", "fieldName3")
                    .addSubAggregation(AggregationBuilders.max("subName1", "fieldName4"))
                    .addSubAggregation(AggregationBuilders.sum("subName2", "fieldName5"))
                    .addSubGroupBy(GroupByBuilders
                        .groupByRange("subName3", "fieldName6")
                        .addRange(12, 90)
                        .addRange(100, 900)
                    ))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the maximum and minimum values for the first level.
    AggregationResults aggResults = resp.getAggregationResults();
    System.out.println(aggResults.getAsMinAggregationResult("name1").getValue());
    System.out.println(aggResults.getAsMaxAggregationResult("name2").getValue());
    // Obtain the GroupByField results of the first level and the results of the aggregatio
ns that are nested in GroupByField.
    GroupByFieldResult results = resp.getGroupByResults().getAsGroupByFieldResult("someName
1");
    for (GroupByFieldResultItem item : results.getGroupByFieldResultItems()) {
        System.out.println("count:" + item.getRowCount());
        System.out.println("key:" + item.getKey());
        // Obtain the sub-aggregation results.
        // Display the maximum value that is obtained from the results of the sub-aggregati
on operation.
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("subNa
me1"));
        // Display the sum that is obtained from the results of the sub-aggregation operati
on.
```

```
        System.out.println(item.getSubAggregationResults().getAsSumAggregationResult("subNa
me2"));
        // Display the GroupByRange values that are obtained from the results of the sub-ag
gregation operation.
        GroupByRangeResult subResults = resp.getGroupByResults().getAsGroupByRangeResult("s
ubName3");
        for (GroupByRangeResultItem subItem : subResults.getGroupByRangeResultItems()) {
            System.out.println("count:" + subItem.getRowCount());
            System.out.println("key:" + subItem.getKey());
        }
    }
}
```

Multiple aggregations

You can perform multiple aggregation operations.

> **Note** If you perform multiple complex aggregation operations at the same time, a long period of time may be required.

- Example 1:

```
public void multipleAggregation(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the minimum value from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").ge
tValue());
    // Obtain the sum from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").ge
tValue());
    // Obtain the number of distinct values from the results of the aggregation operation
.
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
name3").getValue());
}
```

- Example 2

```
public void multipleGroupBy(SyncClient client) {
    // Create a query statement.
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .addGroupBy(GroupByBuilders.groupByField("name4", "type"))
                .addGroupBy(GroupByBuilders.groupByRange("name5", "long").addRange(1, 15)
)
                .build())
        .build();
    // Execute the query statement.
    SearchResponse resp = client.search(searchRequest);
    // Obtain the minimum value from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").ge
tValue());
    // Obtain the sum from the results of the aggregation operation.
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").ge
tValue());
    // Obtain the number of distinct values from the results of the aggregation operation
.
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
name3").getValue());
    // Obtain the values of GroupByField from the results of the aggregation operation.
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("
name4").getGroupByFieldResultItems()) {
        // Display the keys.
        System.out.println(item.getKey());
        // Display the number of rows.
        System.out.println(item.getRowCount());
    }
    // Obtain the values of GroupByRange from the results of the aggregation operation.
    for (GroupByRangeResultItem item : resp.getGroupByResults().getAsGroupByRangeResult("
name5").getGroupByRangeResultItems()) {
        // Display the number of rows.
        System.out.println(item.getRowCount());
    }
}
```

2.8.20. Parallel scan

If you do not have requirements on the order of query results, you can use parallel scan to quickly obtain query results.

Background information

The search index feature allows you to call the Search API operation to query data, sort data in a specific order, and aggregate data.

In some cases, a faster query speed may be more important than the order of query results. For example, when you want to connect Tablestore to a cluster computing environment such as Spark or Presto, or you want to query a specified group of objects. To improve query speeds, Tablestore provides the ParallelScan API operation for the search index feature.

> **Note** Tablestore SDKs 5.6.0 and later versions support the parallel scan feature.

Compared with the Search operation, the ParallelScan operation supports all query features but does not provide analytics capabilities such as sorting and aggregation. This way, query speeds are improved by more than five times. You can call the ParallelScan operation to export hundreds of millions of data rows within a minute. The capability to export data can be horizontally scaled without upper limits.

The maximum number of rows that can be returned by each ParallelScan call is greater than the maximum number of rows that can be returned by each Search call. The Search operation returns up to 100 rows per call, whereas the ParallelScan operation returns up to 2,000 rows per call. The parallel scan feature allows you to use multiple threads to initiate requests in a session in parallel, which accelerates data export.

Scenarios

- If you want to sort or aggregate query results, or the query request is sent from an end user, use the Search operation.
- If you do not need to sort query results and want to quickly return all matched results, or the data is pulled by a computing environment such as Spark or Presto, use the ParallelScan operation.

Features

The differences between the ParallelScan operation and the Search operation lie in the following aspects:

- Stable results

  Parallel scan tasks are stateful. In a session, the result set of the scanned data is determined by the data status when the first request is initiated. If data is inserted or modified after the first request is sent, the result set is not affected.

- Sessions

  Parallel scan-related operations use sessions. The session ID can be used to determine the result set of scanned data. The following process describes how to obtain and use a session ID:

  i. Use the ComputeSplits operation to query the maximum number of parallel scan tasks and the current session ID.

  ii. Initiate multiple parallel scan requests to read data. You must specify the current session ID and the parallel scan task IDs in these requests.

  If the session ID is difficult to obtain, you can call the ParallelScan operation to initiate a request without a specified session ID. However, if you send a request without a specified session ID, there is a very low probability that the obtained result set contains duplicate data.

  Tablestore returns the OTSSessionExpired error code when network exceptions, thread exceptions, dynamic modifications on schemas, or index switchovers occur in the parallel scan process and data scans stop. In these cases, you must initiate another parallel scan task to scan data again.

- Maximum number of parallel scan tasks in a single request

  The maximum number of parallel scan tasks in a single request supported by the ParallelScan operation is determined by the return value of the ComputeSplits request. A larger volume of data requires more parallel scan tasks in a session.

  A single request is specified by one query statement. For example, if you use the Search operation to query results in which the value of city is Hangzhou, all data that matches this condition is returned in the result. However, if you use the ParallelScan operation and the number of parallel scan tasks in a session is 2, each ParallelScan request returns half of the results. The complete result set consists of the two parallel result sets.

- Maximum number of rows that can be returned by each ParallelScan call

  The default value of limit is 2000. The maximum value of limit is 2000. If you enter a value greater than 2000, the performance is not affected.

- Cost

  ParallelScan requests consume fewer resources and are offered at a lower price. To export large amounts of data, we recommend that you use the ParallelScan operation.

- Columns to return

  Only indexed columns can be returned from search indexes. You can set the ReturnType parameter to RETURN_ALL_INDEX or RETURN_SPECIFIED, but not to RETURN_ALL.

  The ParallelScan operation can return only values of the ARRAY and GEOPOINT columns. However, the return values are formatted and may be different from the values that are written to the data table. For example, if you write [1,2, 3, 4] to an ARRAY column, the ParallelScan operation returns [1,2,3,4] as the value. If you write `10,50` to a GEOPOINT column, the ParallelScan operation returns `10.0,50.0` as the value.

- Limits

  The maximum number of parallel scan tasks is 10. You can adjust this limit based on your business requirements. Parallel tasks that have the same session ID and the same ScanQuery parameter value are considered one task. A parallel scan task starts from the time when you send the first ParallelScan request, and ends when all data is scanned or the token expires.

API operations

You can call the following API operations to use the parallel scan feature:

- ComputeSplits: You can call this operation to query the maximum number of parallel scan tasks for a single ParallelScan request.
- ParallelScan: You can call this operation to export data.

Parameters

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

Parameter		Description
scanQuery	query	The query statement for the search index. The operation supports term query, fuzzy query, range query, geo query, and nested query, which are similar to those of the Search operation.
	limit	The maximum number of rows that can be returned by each ParallelScan call.
	maxParallel	The maximum number of parallel scan tasks per request. The maximum number of parallel scan tasks per request varies based on the data volume. A larger volume of data requires more parallel scan tasks per request. You can use the ComputeSplits operation to query the maximum number of parallel scan tasks per request.
	currentParallelId	The ID of the parallel scan task in the request. Valid values: [0, Value of maxParallel)
	token	The token that is used to paginate query results. The results of the ParallelScan request contain the token for the next page. You can use the token to retrieve the next page.
	aliveTime	The validity period of the current parallel scan task. This validity period is also the validity period of the token. Unit: seconds. Default value: 60. We recommend that you use the default value. If the next request is not initiated within the validity period, more data cannot be queried. The validity time of the token is refreshed each time you send a request. **Note** The server uses the asynchronous method to process expired tasks. The current task does not expire within the validity period. However, Tablestore does not guarantee that the task expires after the validity period.
columnsToGet		You can use parallel scan to scan data only in search indexes. To use parallel scan for a search index, you must set store to true when you create the search index.
sessionId		The session ID of the parallel scan task. You can call the ComputeSplits operation to create a session and query the maximum number of parallel scan tasks that are supported by the parallel scan request.

Examples

The following code provides examples on how to scan data by using a single thread or by using multiple threads at the same time:

- Scan data by using a single thread

  When you use parallel scan, the code for a request that uses a single thread is simpler than the code for a request that uses multiple threads. The currentParallelId and maxParallel parameters are not required for a request that uses a single thread. The ParallelScan request that uses a single thread provides higher throughput than the Search request. However, the ParallelScan request that uses a single thread provides lower throughput than the ParallelScan request that uses multiple threads. For more information about how to scan data by using multiple threads at the same time, see the "Scan data by using multiple threads" section.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ParallelScanResponse;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.SearchRequest.ColumnsToGet;
import com.alicloud.openservices.tablestore.model.search.query.MatchAllQuery;
import com.alicloud.openservices.tablestore.model.search.query.Query;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
public class Test {
    public static List<Row> scanQuery(final SyncClient client) {
        String tableName = "<TableName>";
        String indexName = "<IndexName>";
        // Query the session ID and the maximum number of parallel scan tasks supported b
y the request.
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsR
equest);
        byte[] sessionId = computeSplitsResponse.getSessionId();
        int splitsSize = computeSplitsResponse.getSplitsSize();
        /*
         * Create a parallel scan request.
         */
        ParallelScanRequest parallelScanRequest = new ParallelScanRequest();
        parallelScanRequest.setTableName(tableName);
        parallelScanRequest.setIndexName(indexName);
        ScanQuery scanQuery = new ScanQuery();
        // This query determines the range of the data to scan. You can create a nested a
nd complex query.
        Query query = new MatchAllQuery();
        scanQuery.setQuery(query);
        // Specify the maximum number of rows that can be returned by each ParallelScan c
all.
```

```
        scanQuery.setLimit(2000);
        parallelScanRequest.setScanQuery(scanQuery);
        ColumnsToGet columnsToGet = new ColumnsToGet();
        columnsToGet.setColumns(Arrays.asList("col_1", "col_2"));
        parallelScanRequest.setColumnsToGet(columnsToGet);
        parallelScanRequest.setSessionId(sessionId);
        /*
         * Use builder to create a parallel scan request that has the same features as th
e preceding request.
         */
        ParallelScanRequest parallelScanRequestByBuilder = ParallelScanRequest.newBuilder
()
            .tableName(tableName)
            .indexName(indexName)
            .scanQuery(ScanQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(2000)
                .build())
            .addColumnsToGet("col_1", "col_2")
            .sessionId(sessionId)
            .build();
        List<Row> result = new ArrayList<>();
        /*
         * Use the native API operation to scan data.
         */
        {
            ParallelScanResponse parallelScanResponse = client.parallelScan(parallelScanR
equest);
            // Query the token of ScanQuery for the next request.
            byte[] nextToken = parallelScanResponse.getNextToken();
            // Obtain the data.
            List<Row> rows = parallelScanResponse.getRows();
            result.addAll(rows);
            while (nextToken != null) {
                // Specify the token.
                parallelScanRequest.getScanQuery().setToken(nextToken);
                // Continue to scan the data.
                parallelScanResponse = client.parallelScan(parallelScanRequest);
                // Obtain the data.
                rows = parallelScanResponse.getRows();
                result.addAll(rows);
                nextToken = parallelScanResponse.getNextToken();
            }
        }
        /*
         * Recommended method.
         * Use an iterator to scan all matched data. This method has the same query speed
but is easier to use compared with the previous method.
         */
        {
            RowIterator iterator = client.createParallelScanIterator(parallelScanRequestB
yBuilder);
            while (iterator.hasNext()) {
                Row row = iterator.next();
```

```
                result.add(row);
                // Obtain the specific values.
                String col_1 = row.getLatestColumn("col_1").getValue().asString();
                long col_2 = row.getLatestColumn("col_2").getValue().asLong();
            }
        }
        /*
         * If the operation fails, retry the operation. If the caller of this function ha
s a retry mechanism or if you do not want to retry the failed operation, you can ignore t
his part.
         * To ensure availability, we recommend that you start a new parallel scan task w
hen exceptions occur.
         * The following exceptions may occur when you send a ParallelScan request:
         * 1. A session exception occurs on the server side. The error code is OTSSession
Expired.
         * 2. An exception such as a network exception occurs on the client side.
         */
        try {
            // Execute the processing logic.
            {
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequ
estByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    // Process rows of data. If you have sufficient memory resources, you
can add the rows to a list.
                    result.add(row);
                }
            }
        } catch (Exception ex) {
            // Retry the processing logic.
            {
                result.clear();
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequ
estByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    // Process rows of data. If you have sufficient memory resources, you
can add the rows to a list.
                    result.add(row);
                }
            }
        }
        return result;
    }
}
```

- Scan data by using multiple threads

```
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
```

```
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Semaphore;
import java.util.concurrent.atomic.AtomicLong;
public class Test {
    public static void scanQueryWithMultiThread(final SyncClient client, String tableName
, String indexName) throws InterruptedException {
        // Query the number of CPU cores on the client.
        final int cpuProcessors = Runtime.getRuntime().availableProcessors();
        // Specify the number of parallel threads for the client. We recommend that you s
pecify the number of CPU cores on the client as the number of parallel threads for the cl
ient to prevent impact on the query performance.
        final Semaphore semaphore = new Semaphore(cpuProcessors);
        // Query the session ID and the maximum number of parallel scan tasks supported b
y the request.
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsR
equest);
        final byte[] sessionId = computeSplitsResponse.getSessionId();
        final int maxParallel = computeSplitsResponse.getSplitsSize();
        // Create an AtomicLong object if you need to obtain the row count for your busin
ess.
        AtomicLong rowCount = new AtomicLong(0);
        /*
         * If you want to perform multithreading by using a function, you can build an in
ternal class to inherit the threads.
         * You can also build an external class to organize the code.
         */
        final class ThreadForScanQuery extends Thread {
            private final int currentParallelId;
            private ThreadForScanQuery(int currentParallelId) {
                this.currentParallelId = currentParallelId;
                this.setName("ThreadForScanQuery:" + maxParallel + "-" + currentParallelI
d);  // Specify the thread name.
            }
            @Override
            public void run() {
                System.out.println("start thread:" + this.getName());
                try {
                    // Execute the processing logic.
                    {
                        ParallelScanRequest parallelScanRequest = ParallelScanRequest.new
Builder()
                                .tableName(tableName)
                                .indexName(indexName)
                                .scanQuery(ScanQuery.newBuilder()
                                        .query(QueryBuilders.range("col_long").lessThan(1
0_0000)) // Specify the data to query.
                                        .limit(2000)
```

```
                                        .currentParallelId(currentParallelId)
                                        .maxParallel(maxParallel)
                                        .build())
                                .addColumnsToGet("col_long", "col_keyword", "col_bool")
// Specify the fields to return from the search index. To return all fields from the sear
ch index, set returnAllColumnsFromIndex to true.
                                //.returnAllColumnsFromIndex(true)
                                .sessionId(sessionId)
                                .build();
                        // Use an iterator to obtain all the data.
                        RowIterator ltr = client.createParallelScanIterator(parallelScanR
equest);
                        long count = 0;
                        while (ltr.hasNext()) {
                            Row row = ltr.next();
                            // Add a custom processing logic. The following sample code s
hows how to add a custom processing logic to count the number of rows:
                            count++;
                        }
                        rowCount.addAndGet(count);
                        System.out.println("thread[" + this.getName() + "] finished. this
thread get rows:" + count);
                    }
                } catch (Exception ex) {
                    // If exceptions occur, you can retry the processing logic.
                } finally {
                    semaphore.release();
                }
            }
        }
        // Simultaneously execute threads. Valid values of currentParallelId: [0, Value o
f maxParallel).
        List<ThreadForScanQuery> threadList = new ArrayList<ThreadForScanQuery>();
        for (int currentParallelId = 0; currentParallelId < maxParallel; currentParallelI
d++) {
            ThreadForScanQuery thread = new ThreadForScanQuery(currentParallelId);
            threadList.add(thread);
        }
        // Simultaneously initiate the threads.
        for (ThreadForScanQuery thread : threadList) {
            // Specify a value for semaphore to limit the number of threads that can be i
nitiated at the same time to prevent bottlenecks on the client.
            semaphore.acquire();
            thread.start();
        }
        // The main thread is blocked until all threads are complete.
        for (ThreadForScanQuery thread : threadList) {
            thread.join();
        }
        System.out.println("all thread finished! total rows:" + rowCount.get());
    }
}
```

2.8.21. Virtual columns

When you use the virtual column feature, you can modify the schema of a search index or create a search index to query new fields and data of new field types without modifying the storage schema and the data of Tablestore.

Precautions

- The following table describes the data type conversion between virtual columns and columns in data tables.

Field type of data tables	Field type of virtual columns
String	KEYWORD and KEYWORD ARRAY
String	TEXT and TEXT ARRAY
String	LONG and LONG ARRAY
String	DOUBLE and DOUBLE ARRAY
String	GEOPOINT and GEOPOINT ARRAY
Long	Keyword
Long	Text
Double	Keyword
Double	Text

- Virtual columns can be used only in query statements and cannot be used in ColumnsToGet to return column values. To return column values, you can specify that the system returns the source columns of the virtual columns.

Parameters

For more information, see Create search indexes.

Examples

1. Specify a virtual column when you create a search index.
2. Use a virtual column to query data.

   Query the Col_Long_Virtual_Keyword column from a data table whose value can match "1000". Specify that the macthed rows and the total number of matched rows are returned.

```
private static void query(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermsQuery termsQuery = new TermsQuery(); // Set the query type to TermsQuery.
    termsQuery.setFieldName("Col_Long_Virtual_Keyword"); // Set the name of the field t
hat you want to match.
    termsQuery.addTerm(ColumnValue.fromString("1000")); // Set the value that you want
to match.
    searchQuery.setQuery(termsQuery);
    searchQuery.setGetTotalCount(true); // Specify that the total number of matched row
s is returned.
    SearchRequest searchRequest = new SearchRequest("tableName", "indexName", searchQue
ry);
    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setReturnAll(true); // Set ReturnAll to true to return all columns wit
hout returning the virtual column.
    searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount()); // Display the total num
ber of matched columns but not returned rows.
    System.out.println("Row: " + resp.getRows());
}
```

2.9. Secondary Index

2.9.1. Global secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

> **Note** Secondary indexes include global secondary indexes and local secondary indexes. For more information about the secondary index feature, see Secondary index.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created. The value of timeToLive is -1. The value of maxVersions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table on an existing data table.

> **Note** You can create one or more index tables when you create a data table by calling the CreateTable operation. For more information, see Create data tables.

- Parameters

Parameter	Description
mainTableName	The name of the data table.

Parameter	Description
indexMeta	The schema information of the index table. The schema information contains the following items: ◦ indexName: the name of the index table. ◦ primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ◦ definedColumns: the indexed attribute columns. The attribute columns are a combination of predefined columns of the data table. ◦ indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to be indexed and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. ◦ indexUpdateMode: the update mode of the index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ■ If indexUpdateMode is not specified or is set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ■ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the index update mode to IUM_SYNC_INDEX, which indicates the synchronous update mode.

Parameter	Description
includeBaseData	Specifies whether to include the existing data of the data table in the index table. If the last parameter includeBaseData in CreateIndexRequest is set to true, the existing data of the data table is included in the index table. If includeBaseData is set to false, the existing data is excluded.

- Examples

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX2_NAME); // Specify the name of the index ta
ble.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // Specify DEFINED_COL_NAME_1 as t
he first primary key column of the index table.
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2); // Specify PRIMARY_KEY_NAME_2 as t
he second primary key column of the index table.
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); // Specify DEFINED_COL_NAME_2 as an a
ttribute column of the index table.
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); /
/ Create the index table on the data table. Specify that the index table includes the exi
sting data of the data table.
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); //
Create the index table on the data table. Specify that the index table excludes the exist
ing data from the data table.
    /**You can set the IncludeBaseData parameter to true to synchronize existing data fro
m the data table to the index table after the index table is created. Then, you can query
all data from the index table.
       The amount of time required to synchronize data to the index table is determined b
y the amount of data in the data table.
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); // Create the index table.
}
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  When you call the GetRow operation to read data from an index table, take note of the following items:

  - You must set tableName to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from the index table

  For more information, see Multi-row operations.

  - Parameters

    When you call the GetRange operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

  - Examples

    If an index table contains the columns to return, you can query the required data from the index table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("Results returned from the index table:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

If an index table does not contain the columns to return, you must query the required data from the data table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
```

```
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBu
ilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, ke2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); // Specify primary key
columns for the data table based on the primary key columns of the index table.
            // Query the data table.
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, ma
inTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME3); // Specify DEFINED_COL_NAME3 t
o read the DEFINED_COL_NAME3 attribute column from the data table.
            // Set MaxVersions to 1 to read the latest version of data.
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
indexName	The name of the index table.

- Examples

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

2.9.2. Local secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

> **Note** Secondary indexes include global secondary indexes and local secondary indexes. For more information about the secondary index feature, see Secondary index.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created. The value of timeToLive is -1. The value of maxVersions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table on an existing data table.

> **Note** You can create one or more index tables when you create a data table by calling the CreateTable operation. For more information, see Create data tables.

- Parameter

Parameter	Description
mainTableName	The name of the data table.

Parameter	Description
indexMeta	The schema information of the index table. The schema information contains the following items: ◦ indexName: the name of the index table. ◦ primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ◦ definedColumns: the indexed attribute columns. The attribute columns are a combination of predefined columns of the data table. ◦ indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to be indexed and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. ◦ indexUpdateMode: the update mode of the index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ■ If indexUpdateMode is not specified or is set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ■ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the index update mode to IUM_SYNC_INDEX, which indicates the synchronous update mode.

Parameter	Description
includeBaseData	Specifies whether to include the existing data of the data table in the index table. If the last parameter includeBaseData in CreateIndexRequest is set to true, the existing data of the data table is included in the index table. If includeBaseData is set to false, the existing data is excluded.

- Examples

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); // Specify the name of the index tab
le.
    indexMeta.setIndexType(IT_LOCAL_INDEX);     // Set the index type to IT_LOCAL_INDEX,
which specifies a local secondary index.
    indexMeta.setIndexUpdateMode(IUM_SYNC_INDEX);  // Set the index update type to IUM_SY
NC_INDEX, which specifies the synchronous update mode. If the index update type is set to
IT_LOCAL_INDEX, the index update mode must be set to IUM_SYNC_INDEX.
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1); // Add a primary key column to the
index table. The first primary key column of the index table must be the same as the firs
t primary key column of the data table.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_2); // Set DEFINED_COL_NAME_2 to the s
econd primary key column of the index table.
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // Set DEFINED_COL_NAME_1 to the t
hird primary key column of the index table.
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); /
/ Create the index table for the data table. Specify that the index table contains the ex
isting data from the data table.
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); //
Create the index table for the data table. Specify that the index table does not contain
the existing data from the data table.
    /**You can set the IncludeBaseData parameter to true to synchronize existing data fro
m the data table to the index table after the index table is created. Then, you can query
all data from the index table.
       The amount of time required to synchronize data to the index table varies based on
the amount of data in the data table.
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); // Create the index table.
}
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  When you call the GetRow operation to read data from an index table, take note of the following items:

- You must set tableName to the name of the index table.
- Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from the index table

  For more information, see Multi-row operations.

  - Parameters

    When you call the GetRange operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

  - Examples

    If an index table contains the columns to return, you can query the required data from the index table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("Results returned from the index table:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

If the index table does not contain the columns to return, you must query the required data from the data table.

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; // Specify the index table name.
    // Specify the start primary key.
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_1.
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of DEFINED_COL_NAME_1.
```

```
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); // Specify the minimum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // Specify the end primary key.
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of DEFINED_COL_NAME_1.
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); // Specify the maximum value of PRIMARY_KEY_NAME_2.
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBu
ilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, pk2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); // Specify primary key
columns for the data table based on the primary key columns of the index table.
            // Query the data table.
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, ma
inTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME3); // Read the DEFINED_COL_NAME3
attribute column from the data table.
            // Set MaxVersions to 1 to read the latest version of data.
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        // If the nextStartPrimaryKey value is not null, continue to read data.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
indexName	The name of the index table.

- Examples

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

2.10. Tunnel service

2.10.1. Overview

Tablestore SDK for Java encapsulates multiple API operations of Tunnel Service.

The API operations include management operations and the automatic data consumption framework.

- For more information about the management API operations, see the following topics:
  - Create tunnels
  - List the information of a tunnel
  - Query the information of a tunnel
  - Delete tunnels
- For more information about the automatic data consumption framework, see the following topics:
  - Quick start
  - Tunnel clients

2.10.2. Quick start

Tablestore SDK for Java allows you to use Tunnel Service. Before you use Tunnel Service, you must understand the usage notes.

Usage notes

- By default, TunnelWorkerConfig starts the thread pool to read and process data. A single server starts multiple TunnelWorkers. We recommend that TunnelWorkers share the same TunnelWorkerConfig.
- When you create a differential tunnel to consume 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 consume 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 you estimate that the tunnel cannot consume

full data within seven days, submit a ticket or join DingTalk group 23307953 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. The minimum value is 5s.
- When the mode switches from the full channel to the incremental channel, the full channel is closed and the incremental channel is started. This process requires the time for initialization. The heartbeatIntervalInSec parameter determines the initialization time.
- When the TunnelWorker client is abnormally shut down due to an exceptional exit or manual termination, TunnelWorker automatically recycles resources in the following ways: 1. Release the thread pool. 2. Automatically call the shutdown method that you have registered for the Channel class. 3. Shut down the tunnel.

Experience Tunnel Service

Tablestore SDK for Java allows you to use Tunnel Service.

1. Initialize a TunnelClient instance.

```
// Set endPoint to the endpoint of the Tablestore instance. Example: https://instance.cn-hangzhou.ots.aliyuncs.com.
// Set accessKeyId and accessKeySecret to the AccessKey ID and AccessKey secret used to access Tablestore.
// Set instanceName to the name of the instance.
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret, instanceName);
```

2. Create a tunnel.

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

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

3. Customize the data consumption callback function to start automatic data consumption.

```
// Customize the data consumption callback function or call the IChannelProcessor opera
tion. Specify the process and shutdown methods.
private static class SimpleProcessor implements IChannelProcessor {
    @Override
    public void process(ProcessRecordsInput input) {
        // ProcessRecordsInput includes the data that you have obtained.
        System.out.println("Default record processor, would print records count");
        System.out.println(
            // NextToken is used to page the data of Tunnel Client.
            String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
        try {
            // Simulate the processing of data consumption.
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}
// By default, TunnelWorkerConfig starts the thread pool to read and process data. A si
ngle server starts multiple TunnelWorkers.
// We recommend that TunnelWorkers share the same TunnelWorkerConfig. TunnelWorkerConfi
g provides more advanced parameters.
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
// Configure TunnelWorker and start automatic data processing.
TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
try {
    worker.connectAndWorking();
} catch (Exception e) {
    e.printStackTrace();
    worker.shutdown();
    tunnelClient.shutdown();
}
```

Configure TunnelWorkerConfig

TunnelWorkerConfig allows you to customize parameters for a TunnelClient instance based on your requirements. The following table describes the parameters.

Item	Parameter	Description
Configure the interval to detect heartbeats and	heartbeatTimeoutInSec	The timeout period to receive heartbeats. Default value: 300. Unit: seconds. When a heartbeat timeout occurs, the tunnel server considers that the current TunnelClient instance is unavailable. The tunnel client must try to reconnect to the tunnel server.

Item	Parameter	Description
the timeout period to receive heartbeats		
	heartbeatIntervalInSec	The interval to detect heartbeats. You can detect heartbeats to detect active channels, update the status of channels, and automatically initialize data processing tasks. Default value: 30. Minimum value: 5. Unit: seconds.
Interval between checkpoints	checkpointIntervalInMillis	The interval between checkpoints when data is consumed. The interval is recorded on the tunnel server. Default value: 5000. Unit: ms. **Note** • Data to read is stored in different servers. Various errors may occur when you run processes. For example, the server may restart due to environmental factors. The tunnel server regularly records checkpoints after data is processed. A task processes data from the previous checkpoint after the task is restarted. In exceptional conditions, Tunnel Service may sequentially synchronize data once or multiple times. If some data is reprocessed, check the business processing logic. • To prevent data being reprocessed when errors occur, record more checkpoints. However, an excessive number of checkpoints may compromise the system throughput. We recommend that you record the checkpoints based on your business requirements.
The client tag	clientTag	The custom client tag that is used to generate a tunnel client ID. You can customize this parameter to uniquely identify TunnelWorkers.
The custom callback to process data	channelProcessor	The callback that you register to process data, including the process and shutdown methods.
	readRecordsExecutor	The thread pool to read data. If you do not have special requirements, use the default configuration.

Item	Parameter	Description
The configuration of the thread pool to read and process data	processRecordsExecutor	The thread pool to process data. If you do not have special requirements, use the default configuration. **Note** • When you customize the thread pool, we recommend that you set the number of threads to the number of the channels in the tunnel. This way, each channel can be quickly allocated with computing resources such as CPU. • Tablestore performs the following operations for the default configurations of the pool to ensure throughput: ◦ Allocate 32 core threads in advance to guarantee real-time throughput when a small amount of data or a small number of channels exists. ◦ Reduce the queue length when a large amount of data must be processed or when a large number of channels exist. This way, the policy is triggered to create a thread in the pool and quickly allocate more computing resources. ◦ We recommend that you set the thread keep-alive time to 60s. If the amount of data that must be processed is reduced, you can recycle thread resources.
Memory control	maxChannelParallel	The concurrency level of channels to read and process data for memory control. The default value is -1, which indicates that the concurrency level is unlimited. **Note** This configuration applies only to Tablestore SDK for Java V5.10.0 and later.

Item	Parameter	Description
Maximum backoff time	maxRetryIntervalInMillis	The reference value to calculate the maximum backoff time for the tunnel. The maximum backoff time is set to a random value around the reference value. The range of valid values for the maximum backoff time is computed by using the formula: 0.75 × maxRetryIntervalInMillis to 1.25 × maxRetryIntervalInMillis. Default value: 2000 ms. Minimum value: 200 ms. **Note** • This configuration applies only to Tablestore SDK for Java V5.4.0 and later. • If the amount of data that must be processed is smaller than 900 KB or 500 pieces for each export, the tunnel client uses exponential backoff for the tunnel until the maximum backoff time is reached.

References

```
import com.alicloud.openservices.tablestore.TunnelClient;
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelRequest;
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelResponse;
import com.alicloud.openservices.tablestore.model.tunnel.TunnelType;
import com.alicloud.openservices.tablestore.tunnel.worker.IChannelProcessor;
import com.alicloud.openservices.tablestore.tunnel.worker.ProcessRecordsInput;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorker;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorkerConfig;
public class TunnelQuickStart {
    private static class SimpleProcessor implements IChannelProcessor {
        @Override
        public void process(ProcessRecordsInput input) {
            System.out.println("Default record processor, would print records count");
            System.out.println(
                // NextToken is used to page the data of Tunnel Client.
                String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
            try {
                // Simulate the processing of data consumption.
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void shutdown() {
            System.out.println("Mock shutdown");
```

```
        }
    }
    public static void main(String[] args) throws Exception {
        //1. Initialize a TunnelClient instance.
        final String endPoint = "";
        final String accessKeyId = "";
        final String accessKeySecret = "";
        final String instanceName = "";
        TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret
, instanceName);
        //2. Create a tunnel. You must use the createTable method in SyncClient or go to th
e Tablestore console to create a table for testing in advance.
        final String tableName = "testTable";
        final String tunnelName = "testTunnel";
        CreateTunnelRequest request = new CreateTunnelRequest(tableName, tunnelName, Tunnel
Type.BaseAndStream);
        CreateTunnelResponse resp = tunnelClient.createTunnel(request);
        // Use tunnelId to initialize TunnelWorker. You can call the ListTunnel or Describe
Tunnel operation to obtain the tunnel ID.
        String tunnelId = resp.getTunnelId();
        System.out.println("Create Tunnel, Id: " + tunnelId);
        //3. Customize the data consumption callback to start automatic data consumption.
        // TunnelWorkerConfig contains more advanced parameters.
        TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
        TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
        try {
            worker.connectAndWorking();
        } catch (Exception e) {
            e.printStackTrace();
            worker.shutdown();
            tunnelClient.shutdown();
        }
    }
}
```

2.10.3. Create tunnels

You can call the CreateTunnel operation to create one or more tunnels for a table. When you create a tunnel, you must specify the table name, tunnel name, and tunnel type.

Request parameters

Parameter	Description
tableName	The name of the table for which you want to create a tunnel.
tunnelName	The tunnel name.

Parameter	Description
tunnelType	The tunnel type. Valid values: • BaseData: Only full data is consumed and processed. • Stream: Only incremental data is consumed and processed. • BaseAndStream: After full data is consumed and processed, incremental data is consumed and processed.

Response parameters

Parameter	Description
TunnelId	The tunnel ID.
ResponseInfo	Some other fields returned.
RequestId	The ID of the request.

Examples

- Example 1

  The following code provides an example on how to create a tunnel of the BaseData type:

```
// You can create three types of tunnels: TunnelType.BaseData, TunnelType.Stream, and Tun
nelType.BaseAndStream.
// In this example, a BaseData tunnel is created. To create other types of tunnels, set T
unnelType in CreateTunnelRequest to the required types.
private static void createTunnel(TunnelClient client, String tunnelName) {
    CreateTunnelRequest request = new CreateTunnelRequest(TableName, tunnelName, TunnelTy
pe.BaseData);
    CreateTunnelResponse resp = client.createTunnel(request);
    System.out.println("RequestId: " + resp.getRequestId());
    System.out.println("TunnelId: " + resp.getTunnelId());
}
```

- Example 2

  The following code provides an example on how to create a tunnel of the Stream or BaseAndStream type and how to specify the time range to read incremental data:

```
// Create a tunnel of the Stream or BaseAndStream type. Specify the start timestamp or en
d timestamp to read incremental data within a specified time range. The configurations sp
ecified by StreamTunnelConfig do not take effect for BaseData tunnels.
private static void createStreamTunnelByOffset(TunnelClient client,String tableName,Strin
g tunnelName){
    CreateTunnelRequest createTunnelRequest = new CreateTunnelRequest(tableName,tunnelNam
e, TunnelType.Stream);// Create a tunnel of the Stream type.
    //CreateTunnelRequest createTunnelRequest = new CreateTunnelRequest(tableName,tunnelN
ame, TunnelType.BaseAndStream);// Create a tunnel of the BaseAndStream type.
    StreamTunnelConfig streamTunnelConfig = new StreamTunnelConfig();
    /*
        Specify the start timestamp and end timestamp to read incremental data. Unit: mil
liseconds. Valid values: [CurrentSystemTime - StreamExpiration + 5 minute, CurrentSystemT
ime).
        CurrentSystemTime is a timestamp that indicates the current system time in millis
econds. StreamExpiration is a timestamp of the time when the incremental logs expire in m
illiseconds. The maximum validity period of incremental logs is seven days. You can set S
treamExpiration in the attributes of the table.
        The value of endTime must be greater than that of startTime.
     */
    streamTunnelConfig.setStartOffset(startTime);
    streamTunnelConfig.setEndOffset(endTime);
    createTunnelRequest.setStreamTunnelConfig(streamTunnelConfig);
    CreateTunnelResponse resp = client.createTunnel(createTunnelRequest);
    System.out.println("RequestId: " + resp.getRequestId());
    System.out.println("TunnelId: " + resp.getTunnelId());
}
```

2.10.4. List the information of a tunnel

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

Request parameters

Parameter	Description
TableName	The name of the table that you want to list tunnels.

Response parameters

Parameter	Description

Parameter	Description
List<TunnelInfo>	The list of the tunnel, including the following information: • TunnelId: the ID of the tunnel. • TunnelType: the type of the tunnel. Valid values: BaseData, Stream, and BaseAndStream. • TableName: the name of the table for which the tunnel is configured. • InstanceName: the name of the instance for which the tunnel is configured. • Stage: the stage at which the tunnel is located. Valid values: InitBaseDataAndStreamShard, ProcessBaseData, and ProcessStream. • Expired: indicates whether the data is expired. If true is returned, contact Tablestore technical support by using DingTalk in a timely manner.
ResponseInfo	Some other fields returned in the request.
RequestId	The ID of the request.

Examples

```
private static void listTunnel(TunnelClient client, String tableName) {
    ListTunnelRequest request = new ListTunnelRequest(tableName);
    ListTunnelResponse resp = client.listTunnel(request);
    System.out.println("RequestId: " + resp.getRequestId());
    for (TunnelInfo info : resp.getTunnelInfos()) {
        System.out.println("TunnelInfo::::::");
        System.out.println("\tTunnelName: " + info.getTunnelName());
        System.out.println("\tTunnelId: " + info.getTunnelId());
        // The type of the tunnel. Valid values: BaseData, Stream, and BaseAndStream.
        System.out.println("\tTunnelType: " + info.getTunnelType());
        System.out.println("\tTableName: " + info.getTableName());
        System.out.println("\tInstanceName: " + info.getInstanceName());
        // The stage at which the tunnel is located. Valid values: InitBaseDataAndStreamSha
rd, ProcessBaseData, and ProcessStream.
        System.out.println("\tStage: " + info.getStage());
        // Indicate whether data is expired. If true is returned, contact Tablestore techni
cal support by using DingTalk in a timely manner.
        System.out.println("\tExpired: " + info.isExpired());
    }
}
```

2.10.5. Query the information of a tunnel

You can call the DescribeTunnel operation to obtain the information of channels in a specified tunnel. A channel corresponds to a shard for the Stream operation of Tablestore.

Request parameters

Parameter	Description
TableName	The name of the table for which you want to query the tunnel information.
TunnelName	The tunnel name.

Response parameters

Parameter	Description
TunnelConsumePoint	The last time when the tunnel consumes incremental data. The time equals the point in time when the slowest channel in the tunnel consumes data. Default value: January 1, 1970 (UTC).
TunnelInfo	The list of the tunnel, including the following information: • TunnelId: the ID of the tunnel. • TunnelType: the type of the tunnel. Valid values: BaseData, Stream, and BaseAndStream. • TableName: the name of the table for which the tunnel is configured. • InstanceName: the name of the instance for which the tunnel is configured. • Stage: the stage at which the tunnel is located. Valid values: InitBaseDataAndStreamShard, ProcessBaseData, and ProcessStream. • Expired: indicates whether the data is expired. If true is returned, contact Tablestore technical support by using DingTalk in a timely manner.

Parameter	Description
List<ChannelInfo>	The list of channels in the tunnel, including the following information: • ChannelId: the ID of the channel. • ChannelType: the type of the channel. Valid values: Base and Stream. • ChannelStatus: the status of the channel. Valid values: WAIT, OPEN, CLOSING, CLOSE, and TERMINATED. • ClientId: the ID of the tunnel client. By default, the client ID concatenates the hostname of the client and a random string. The client hostname can be customized by TunnelWorkerConfig. • ChannelConsumePoint: the last time when a channel consumes incremental data. Default value: January 1, 1970 (UTC). This parameter is unavailable for the full data consumption. • ChannelCount: the number of data entries that a channel synchronizes.
ResponseInfo	Some other fields returned in the request.
RequestId	The ID of the request.

Examples

```
// 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 last time when the tunnel consumes incremental data. The time equals the point i
n time when the slowest channel in the tunnel consumes data. Default value: January 1, 1970
(UTC).
    System.out.println("TunnelConsumePoint: " + resp.getTunnelConsumePoint());
    System.out.println("TunnelInfo: " + resp.getTunnelInfo());
    for (ChannelInfo ci : resp.getChannelInfos()) {
        System.out.println("ChannelInfo::::::");
        System.out.println("\tChannelId: " + ci.getChannelId());
        // 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 client ID concatenates the hostname
of the client and a random string.
        System.out.println("\tClientId: " + ci.getClientId());
        // The last time when the channel consumes incremental data.
        System.out.println("\tChannelConsumePoint: " + ci.getChannelConsumePoint());
        // The number of data entries that the channel synchronizes.
        System.out.println("\tChannelCount: " + ci.getChannelCount());
    }
}
```

2.10.6. Delete tunnels

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

Request parameters

Parameter	Description
TableName	The name of the table for which you want to delete the tunnel.
TunnelName	The tunnel name.

Response parameters

Parameter	Description
ResponseInfo	Some other fields returned.
RequestId	The ID of the request.

Examples

```
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());
}
```

2.11. SQL query

2.11.1. Overview

This topic describes the operations that you can perform when you use Tablestore SDK for Java to execute SQL statements.

> **Notice** Tablestore SDK for Java V5.13.0 or later supports the SQL query feature. Before you use the SQL query feature, make sure that Tablestore SDK for Java V5.13.0 or later is obtained. For more information about the version history of Tablestore SDK for Java, see Tablestore SDK for Java.

Operation	Description
Create mapping tables for tables	Execute the create table statement to create tables and mapping tables. If a table with the specified name exists, only the mapping table is created. If a table with the specified name does not exist, both the table and the mapping table are created. **Notice** The create table statement creates mapping tables for existing tables but cannot create tables.
Delete mapping tables	Execute the drop mapping table statement to delete mapping tables.
List the names of tables	Execute the show tables statement to query the names of tables in the current database.
Query information about tables	Execute the describe statement to query the information about tables, such as the field names and field types.
Query the index information about tables	Execute the show index statement to query the index information about tables.
Query data in tables	Execute the select statement to query data in tables.

2.11.2. Create mapping tables for tables

You can execute the create table statement to create a mapping table for an existing table or search index.

> **Note** For more information about the create table statement, see Create mapping tables for tables and Create mapping tables for search indexes.

Prerequisites

A Tablestore client is initialized. For more information, see Initialization.

Usage notes

Tablestore SDK for Java V5.13.0 or later supports the SQL query feature. Before you use the SQL query feature, make sure that Tablestore SDK for Java V5.13.0 or later is obtained. For more information about the version history of Tablestore SDK for Java, see Tablestore SDK for Java.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Example

> **Note** The following code provides an example on how to create a mapping table for a table.

Execute the `create table test_table (pk varchar(1024), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primary key(pk))` statement to create a mapping table for the table named test_table.

```
private static void createTable(SyncClient client) {
    // Create a SQL request.
    SQLQueryRequest request = new SQLQueryRequest("create table test_table (pk varchar(1024
), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primar
y key(pk))");
    // Obtain the response to the SQL request.
    SQLQueryResponse response = client.sqlQuery(request);
}
```

2.11.3. Delete mapping tables

You can execute the drop mapping table statement to delete mapping tables.

> **Note** For more information about the drop mapping table statement, see Delete mapping tables.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Usage notes

Tablestore SDK for Java V5.13.0 or later supports the SQL query feature. Before you use the SQL query feature, make sure that Tablestore SDK for Java V5.13.0 or later is obtained. For more information about the version history of Tablestore SDK for Java, see Tablestore SDK for Java.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Example

Execute the drop mapping table test_table statement to delete the mapping table of the table named test_table.

```
private static void dropMappingTable(SyncClient client) {
    // Create a SQL request.
    SQLQueryRequest request = new SQLQueryRequest("drop mapping table test_table");
    // Obtain the response to the SQL request.
    SQLQueryResponse response = client.sqlQuery(request);
}
```

2.11.4. List the names of tables

You can execute the show tables statement to list the names of tables in the current database.

> **Note** For more information about the show tables statement, see List table names.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Usage notes

Tablestore SDK for Java V5.13.0 or later supports the SQL query feature. Before you use the SQL query feature, make sure that Tablestore SDK for Java V5.13.0 or later is obtained. For more information about the version history of Tablestore SDK for Java, see Tablestore SDK for Java.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Example

Execute the `show tables` statement to list the names of tables.

```
private static void showTable(SyncClient client) {
    // Create a SQL request.
    SQLQueryRequest request = new SQLQueryRequest("show tables");
    // Obtain the response to the SQL request.
    SQLQueryResponse response = client.sqlQuery(request);
    // Obtain the schema of the returned results of the SQL query statement.
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table schema: " + tableMeta.getSchema());
    // Use SQL ResultSet to obtain all returned results of the SQL query statement.
    System.out.println("response resultset:");
    SQLResultSet resultSet = response.getSQLResultSet();
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0));
    }
    // Use the SQLUtils function to parse the response to obtain the list of table names.
    System.out.println("response sqlutils resultset:");
    List<String> tables = SQLUtils.parseShowTablesResponse(response);
    for (String table : tables) {
        System.out.print(table + ", ");
    }
}
```

Sample output:

```
response table schema: [Tables_in_$instanceName:STRING]
response resultset:
test_table
response sqlutils resultset:
test_table,
```

2.11.5. Query information about tables

You can execute the describe statement to query information about tables, such as the field names and field types.

> **Note** For more information about the describe statement, see Query the information about a table.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Usage notes

Tablestore SDK for Java V5.13.0 or later supports the SQL query feature. Before you use the SQL query feature, make sure that Tablestore SDK for Java V5.13.0 or later is obtained. For more information about the version history of Tablestore SDK for Java, see Tablestore SDK for Java.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Example

Execute the `describe test_table` statement to query information about the table named test_table.

```
private static void getTableDesc(SyncClient client) {
    // Create a SQL request.
    SQLQueryRequest request = new SQLQueryRequest("describe test_table");
    // Obtain the response to the SQL request.
    SQLQueryResponse response = client.sqlQuery(request);
    // Obtain the schema of the returned results of the SQL query statement.
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table schema: " + tableMeta.getSchema());
    // Use SQL ResultSet to obtain all returned results of the SQL query statement.
    System.out.println("response resultset:");
    SQLResultSet resultSet = response.getSQLResultSet();
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0) + ", " + row.getString(1) + ", " +
                           row.getString(2) + ", " + row.getString(3) + ", " +
                           row.getString(4) + ", " + row.getString(5));
    }
}
```

Sample output:

```
response table schema: [Field:STRING, Type:STRING, Null:STRING, Key:STRING, Default:STRING,
Extra:STRING]
response resultset:
pk, varchar(1024), NO, PRI, null,
long_value, bigint(20), YES, , null,
double_value, double, YES, , null,
string_value, mediumtext, YES, , null,
bool_value, tinyint(1), YES, , null,
```

2.11.6. Query the index information about tables

You can execute the show index statement to query the index information about tables.

> **Note** For more information about the show index statement, see Query the index information about a table.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Usage notes

Tablestore SDK for Java V5.13.0 or later supports the SQL query feature. Before you use the SQL query feature, make sure that Tablestore SDK for Java V5.13.0 or later is obtained. For more information about the version history of Tablestore SDK for Java, see Tablestore SDK for Java.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `show index in test_table` statement to query the index information about the table named test_table.

```
private static void showIndexDemo(SyncClient client) {
    // Create a SQL request.
    SQLQueryRequest request = new SQLQueryRequest("show index in test_table");
    // Obtain the response to the SQL request.
    SQLQueryResponse response = client.sqlQuery(request);
    // Obtain the schema of the returned results of the SQL query statement.
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table schema: " + tableMeta.getSchema());
    // Use SQL ResultSet to obtain all returned results of the SQL query statement.
    System.out.println("response resultset:");
    SQLResultSet resultSet = response.getSQLResultSet();
    while (resultSet.hasNext()) {
        SQLRow row = showIndexResultSet.next();
        System.out.println(row.getString("Table") + ", " + row.getLong("Non_unique") + ", "
+
                           row.getString("Key_name") + ", " + row.getLong("Seq_in_index") +
", " +
                           row.getString("Column_name") + ", " + row.getString("Index_type"
) );
    }
}
```

Sample output:

```
response table schema: [Table:STRING, Non_unique:INTEGER, Key_name:STRING, Seq_in_index:INT
EGER, Column_name:STRING, Is_defined_column:STRING, Collation:STRING, Cardinality:INTEGER,
Sub_part:INTEGER, Packed:STRING, Null:STRING, Index_type:STRING, Comment:STRING, Index_comm
ent:STRING, Visible:STRING, Expression:STRING]
response resultset:
test_table, 0, PRIMARY, 1, pk,
test_table, 1, test_table_index, 1, pk, SearchIndex
test_table, 1, test_table_index, 2, bool_value, SearchIndex
test_table, 1, test_table_index, 3, double_value, SearchIndex
test_table, 1, test_table_index, 4, long_value, SearchIndex
test_table, 1, test_table_index, 5, string_value, SearchIndex
```

2.11.7. Query data in tables

You can execute the select statement to query data in tables.

> **Note** For more information about the select statement, see Query data.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Usage notes

Tablestore SDK for Java V5.13.0 or later supports the SQL query feature. Before you use the SQL query feature, make sure that Tablestore SDK for Java V5.13.0 or later is obtained. For more information about the version history of Tablestore SDK for Java, see Tablestore SDK for Java.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `select pk, long_value, double_value, string_value, bool_value from test_table limit 20` statement to query data in the table named test_table and set the maximum number of rows that you want to return to 20. The system returns the request type, the schema of the returned results, and the returned results of the query statement.

```
private static void queryData(SyncClient client) {
    // Create a SQL request.
    SQLQueryRequest request = new SQLQueryRequest("select pk, long_value, double_value, str
ing_value, bool_value from test_table limit 20");
    // Obtain the response to the SQL request.
    SQLQueryResponse response = client.sqlQuery(request);
    // Obtain the SQL request type.
    System.out.println("response type: " + response.getSQLStatementType());
    // Obtain the schema of the returned results of the SQL request.
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table meta: " + tableMeta.getSchema());
    // Obtain the returned results of the SQL request.
    SQLResultSet resultSet = response.getSQLResultSet();
    System.out.println("response resultset:");
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0) + ", " + row.getString("pk") + ", " +
                           row.getLong(1) + ", " + row.getLong("long_value") + ", " +
                           row.getDouble(2) + ", " + row.getDouble("double_value") + ", " +
                           row.getString(3) + ", " + row.getString("string_value") + ", " +
                           row.getBoolean(4) + ", " + row.getBoolean("bool_value"));
    }
}
```

Sample output:

```
response type: SQL_SELECT
response table meta: [pk:STRING, long_value:INTEGER, double_value:DOUBLE, string_value:STRI
NG, bool_value:BOOLEAN]
response resultset:
binary_null, binary_null, 1, 1, 1.0, 1.0, a, a, false, false
bool_null, bool_null, 1, 1, 1.0, 1.0, a, a, null, null
double_null, double_null, 1, 1, null, null, a, a, true, true
long_null, long_null, null, null, 1.0, 1.0, a, a, true, true
string_null, string_null, 1, 1, 1.0, 1.0, null, null, false, false
```

2.12. Data Delivery

2.12.1. Create delivery tasks

You can call the CreateDeliveryTask operation to create a delivery task. You can create a delivery task to deliver data from Tablestore to an OSS bucket.

> **Notice**
>
> Make sure that Tablestore SDK for Java is installed to support delivery service. For more information about Tablestore SDK for Java, see Tablestore SDK for Java.

Prerequisites

- OSS is activated. A bucket is created in the region where a Tablestore instance is deployed. For more information, see Activate OSS.
- The Tablestore service linked role (AliyunServiceRoleForOTSDataDelivery) is created in the console. The Alibaba Cloud Resource Name (ARN) of the role is recorded. For more information, see the "Create delivery tasks" section of the Quick start topic.

  You can obtain the ARN of the Tablestore service linked role (AliyunServiceRoleForOTSDataDelivery) in the RAM console. Perform the following operations:

  On the **RAM Roles** page, search AliyunServiceRoleForOTSDataDelivery, and then click the RAM role name. On the role details page, you can view and copy the ARN information of the role.
- The OTSClient instance is initialized. For more information, see Initialization.
- A table is created. Data is written to the table.

Parameters

Parameter	Description
tableName	The name of the table.
taskName	The name of the delivery task. The name must be 3 to 16 characters in length and can contain only lowercase letters, digits, and hyphens (-). It must start and end with a lowercase letter or digit.

Parameter	Description

taskConfig	The configurations of the delivery task. Valid values:
	• ossPrefix: the prefix of the folder in the bucket. Data is delivered from Tablestore to the folder. The path of the destination folder supports the following time variables: $yyyy, $MM, $dd, $HH, and $mm.
	  ◦ When the path uses time variables for delivery, OSS folders are dynamically generated based on the time when data is written. This way, data is partitioned based on the naming conventions that are followed when Hive partitions data. Objects in OSS are organized, partitioned, and distributed based on time.
	  ◦ When the path does not use time variables, all files are delivered to an OSS folder whose name contains a specific prefix.
	• ossBucket: the name of the OSS bucket.
	• ossEndpoint: the endpoint of the region where an OSS bucket is located.
	• ossStsRole: the Alibaba Cloud Resource Name (ARN) of the Tablestore service-linked role.
	• format: the format of the delivered data. The delivered data is stored in the Parquet format. By default, data delivery uses PLAIN to encode data of any type.
	• eventTimeColumn: the event time column. This parameter specifies that data is partitioned based on the time of data in a column. If you do not specify this parameter, data is partitioned based on the time when the data is written to Tablestore.
	• parquetSchema: specifies the column you want to deliver. You must configure the source fields, destination fields, and destination field types to deliver.
	The order in which fields are sorted in Tablestore can be different from the order of fields in the schema. Parquet data stored in OSS is distributed based on the order of fields in the schema.
	Notice
	The data types must be consistent between the source and destination fields. If the data types between the fields are inconsistent, the fields are discarded as dirty data. For more information about field type mappings, see the "Data type mappings" section of the Quick start topic.

Parameter	Description
taskType	The mode in which to deliver data. Default value: BASE_INC. Valid values: • INC: the incremental data delivery mode. Only incremental data is synchronized. • BASE: the full data delivery mode. All data in tables is scanned and synchronized. • BASE_INC: the differential data delivery mode. After the full data is synchronized, Tablestore synchronizes the incremental data. When you synchronize data in incremental mode, you can view the time when data is last delivered and the status of the current delivery task.

Examples

```
private static void createDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    String taskName = "sampledeliverytask";
    OSSTaskConfig taskConfig = new OSSTaskConfig();
    taskConfig.setOssPrefix("sampledeliverytask/year=$yyyy/month=$MM");
    taskConfig.setOssBucket("datadeliverytest");
    taskConfig.setOssEndpoint("oss-cn-hangzhou.aliyuncs.com");
    taskConfig.setOssStsRole("acs:ram::17 %%%)%%%%*****: role/aliyunserviceroleforotsdatade
livery"); // eventColumn is optional. eventColumn specifies that data is partitioned based
on the time of a column. If you do not specify this parameter, data is partitioned based on
the time when the data is written to Tablestore.
    EventColumn eventColumn = new EventColumn("PK1", EventTimeFormat.RFC1123);
    taskConfig.setEventTimeColumn(eventColumn);
    taskConfig.addParquetSchema(new ParquetSchema("PK1", "PK1", DataType.UTF8));
    taskConfig.addParquetSchema(new ParquetSchema("PK2", "PK2", DataType.BOOL));
    taskConfig.addParquetSchema(new ParquetSchema("Col1", "Col1", DataType.UTF8));
    CreateDeliveryTaskRequest request = new CreateDeliveryTaskRequest();
    request.setTableName(tableName);
    request.setTaskName(taskName);
    request.setTaskConfig(taskConfig);
    request.setTaskType(DeliveryTaskType.BASE_INC);
    CreateDeliveryTaskResponse response = client.createDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    System.out.println("create delivery task success");
}
```

2.12.2. List the names of delivery tasks

You can call the ListDeliveryTask operation to list information about all deliver tasks for a table.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.

- The delivery task is created. For more information about how to use the console or the SDK, see Quick start or Create delivery tasks.

Request parameters

Parameter	Description
tableName	The name of the table.

Response parameters

Parameter	Description
tableName	The name of the table, which is the same as the name of the table in the request.
taskName	The name of the delivery task.
taskType	The type of the delivery task.

Examples

```
public static void listDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    ListDeliveryTaskRequest request = new ListDeliveryTaskRequest(tableName);
    ListDeliveryTaskResponse response = client.listDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    for(DeliveryTaskInfo taskInfo: response.getTaskInfos()) {
        System.out.println("tableName: " + taskInfo.getTableName());
        System.out.println("taskName: " + taskInfo.getTaskName());
        System.out.println("taskType: " + taskInfo.getTaskType().toString());
    }
}
```

2.12.3. Query the descriptive information of delivery tasks

You can call the DescribeDeliveryTask operation to query the descriptive information of a delivery task.

Request parameters

Parameter	Description
tableName	The name of the table.
taskName	The name of the delivery task.

Response parameters

Parameter	Description
taskConfig	The configuration information of the delivery task.
taskSyncStat	The synchronization status of the delivery task.
taskType	The type of the delivery task.

Examples

```
public static void describeDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    String taskName  = "sampledeliverytask";
    DescribeDeliveryTaskRequest request = new DescribeDeliveryTaskRequest(tableName, taskNa
me);
    DescribeDeliveryTaskResponse response = client.describeDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    System.out.println("OSSconfig: " + response.getTaskConfig());
    System.out.println("TaskSyncStat: " + response.getTaskSyncStat());
    System.out.println("taskType: " + response.getTaskType());
}
```

2.12.4. Delete delivery tasks

You can call the DeleteDeliveryTask operation to delete a delivery task.

Parameters

Parameter	Description
tableName	The name of the table.

Parameter	Description
taskName	The name of the delivery task.

Examples

```
private static void deleteDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    String taskName = "sampledeliverytask";
    DeleteDeliveryTaskRequest request = new DeleteDeliveryTaskRequest(tableName, taskName);
    DeleteDeliveryTaskResponse response = client.deleteDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    System.out.println("delete task delivery success");
}
```

2.13. TimeSeries model

2.13.1. Overview

Tablestore SDK for Java provides a variety of operations that you can perform on time series tables.

Operation	Description
Create a time series table	Creates a time series table.
Query the names of time series tables	Queries the names of all time series tables in the current instance.
Query the information about a time series table	Queries the information about a time series table, such as the time to live (TTL) configuration.
Update a time series table	Updates the configurations of a time series table.
Delete a time series table	Deletes a time series table.
Write time series data	Writes multiple rows of time series data to a time series table at the same time.
Query time series data	Queries the time series data that meets the specified conditions.
Retrieve time series	Retrieves time series that meet multiple specified conditions.
Update time series metadata	Updates the properties in the metadata of multiple time series at the same time.
Delete time series metadata	Deletes the metadata of multiple time series at the same time.

2.13.2. Create a time series table

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

Prerequisites

- An instance is created in the Tablestore console. For more information, see Create an instance for the TimeSeries model.
- The TimeseriesClient is initialized. For more information, see Initialization.

Usage note

The name of the time series table that you create cannot be the same as the name of an existing data table.

Parameters

Example

Create a time series table named test_timeseries_table and set the data in the time series table to never expire.

2.13.3. Query the names of time series tables

You can call the ListTimeseriesTable operation to query the names of all time series tables in the current instance.

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Example

Query the names of all time series tables in an instance.

```
private static void listTimeseriesTable(TimeseriesClient client) {
    ListTimeseriesTableResponse listTimeseriesTableResponse = client.listTimeseriesTable();
    for (String table : listTimeseriesTableResponse.getTimeseriesTableNames()) {
        System.out.println(table);
    }
}
```

2.13.4. Query the information about a time series table

You can call the DescribeTimeseriesTable operation to query the information about a time series table, such as the time to live (TTL) configuration.

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameter

Parameter	Description
timeseriesTableName	The name of the time series table.

Example

Query the information about a time series table named test_timeseries_table.

```
private static void describeTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    DescribeTimeseriesTableResponse describeTimeseriesTableResponse = client.describeTimese
riesTable(new DescribeTimeseriesTableRequest(tableName));
    TimeseriesTableMeta tableMeta = describeTimeseriesTableResponse.getTimeseriesTableMeta(
);
    System.out.println(tableMeta.getTimeseriesTableName()); // The name of the time series
table.
    System.out.println(tableMeta.getStatus()); // The status of the time series table.
    System.out.println(tableMeta.getTimeseriesTableOptions().getTimeToLive()); // The TTL c
onfiguration of the time series table.
}
```

2.13.5. Update a time series table

You can call the UpdateTimeseriesTable operation to update the configurations of a time series table, such as the time to live (TTL).

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

For more information, see Create a time series table.

Example

Update the TTL of the data in a time series table named test_timeseries_table to three years.

```
private static void updateTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    UpdateTimeseriesTableRequest updateTimeseriesTableRequest = new UpdateTimeseriesTableRe
quest(tableName);
    updateTimeseriesTableRequest.setTimeseriesTableOptions(new TimeseriesTableOptions(86400
* 365 * 3)); // Update the TTL of the data in the time series table to three years.
    client.updateTimeseriesTable(updateTimeseriesTableRequest);
    DescribeTimeseriesTableResponse describeTimeseriesTableResponse = client.describeTimese
riesTable(new DescribeTimeseriesTableRequest(tableName));
    TimeseriesTableMeta tableMeta = describeTimeseriesTableResponse.getTimeseriesTableMeta(
);
    System.out.println(tableMeta.getTimeseriesTableOptions().getTimeToLive()); // View the
updated TTL configuration of the time series table.
}
```

2.13.6. Delete a time series table

You can call the DeleteTimeseries operation to delete a time series table.

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameter

Parameter	Description
timeseriesTableName	The name of the time series table.

Example

Delete a time series table named test_timeseries_table.

```
private static void deleteTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    DeleteTimeseriesTableRequest deleteTimeseriesTableRequest = new DeleteTimeseriesTableRe
quest(tableName);
    client.deleteTimeseriesTable(deleteTimeseriesTableRequest);
}
```

2.13.7. Write time series data

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

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

Example

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

2.13.8. Query time series data

You can call the GetTimeseriesData operation to query the time series data that meets the specified conditions in a time series table.

Prerequisites

- Time series data is written to the time series table. For more information, see Write time series data.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

Example

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

2.13.9. Retrieve time series

You can call the QueryTimeseriesMeta operation to retrieve time series that meet multiple specified conditions.

Prerequisites

- Time series data is written to the time series table. For more information, see Write time series data.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

Example

Query all time series in which the metric name is cpu and the tags contain the os tag that is prefixed with Ubuntu in a time series table named test_timeseries_table.

2.13.10. Update time series metadata

You can call the UpdateTimeseriesMeta operation to update the metadata of multiple time series at the same time.

Prerequisites

- Time series data is written to the time series table. For more information, see Write time series data.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

The timeseriesMeta parameter specifies the metadata of a time series. Each timeseriesMeta parameter consists of the timeseriesKey and attributes parameters. The following table describes the parameters.

Parameter	Description
timeseriesKey	The identifier of the time series.
attributes	The properties of the time series. The value is a key-value pair of the string type.

Example

Update the properties of the time series that meet the specified conditions in a time series table named test_timeseries_table.

```
private static void updateTimeseriesMeta(TimeseriesClient client) {
    List<TimeseriesMeta> timeseriesMetaList = new ArrayList<TimeseriesMeta>();
    for (int i = 0; i < 10; i++) {
        Map<String, String> tags = new HashMap<String, String>();
        tags.put("region", "hangzhou");
        tags.put("os", "Ubuntu16.04");
        // Construct TimeseriesKey.
        TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_" + i, tags);
        TimeseriesMeta meta = new TimeseriesMeta(timeseriesKey);
        // Specify the properties of the time series.
        Map<String, String> attrs = new HashMap<String, String>();
        attrs.put("status", "online");
        meta.setAttributes(attrs);
        timeseriesMetaList.add(meta);
    }
    String tableName = "test_timeseries_table";
    UpdateTimeseriesMetaRequest updateTimeseriesMetaRequest = new UpdateTimeseriesMetaReque
st(tableName);
    updateTimeseriesMetaRequest.setMetas(timeseriesMetaList);
    UpdateTimeseriesMetaResponse updateTimeseriesMetaResponse = client.updateTimeseriesMeta
(updateTimeseriesMetaRequest);
    // Check whether the properties of the time series are updated.
    if (!updateTimeseriesMetaResponse.isAllSuccess()) {
        for (UpdateTimeseriesMetaResponse.FailedRowResult failedRowResult : updateTimeserie
sMetaResponse.getFailedRows()) {
            System.out.println(failedRowResult.getIndex());
            System.out.println(failedRowResult.getError());
        }
    }
}
```

2.13.11. Delete time series metadata

You can call the DeleteTimeseriesMeta operation to delete the metadata of multiple time series at a time.

Prerequisites

- Time series data is written to the time series table. For more information, see Write time series data.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

A time series identifier is used to identify a time series. You can specify multiple time series identifiers to delete the metadata of multiple time series. The following table describes the parameter.

Parameter	Description
timeseriesKey	The identifier of the time series. The identifier includes the following content: • measurementName: the measurement name of the time series. • dataSource: the data source of the time series. You can leave this parameter empty. • tags: the tags of the time series. The tags are multiple key-value pairs of the STRING type.

Example

Delete the metadata of the time series that meet the specified conditions from the time series table named test_timeseries_table.

```
private static void deleteTimeseriesMeta(TimeseriesClient client) {
    List<TimeseriesKey> timeseriesKeyList = new ArrayList<TimeseriesKey>();
    for (int i = 0; i < 10; i++) {
        Map<String, String> tags = new HashMap<String, String>();
        tags.put("region", "hangzhou");
        tags.put("os", "Ubuntu16.04");
        // Construct a time series identifier.
        TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_" + i, tags);
        timeseriesKeyList.add(timeseriesKey);
    }
    String tableName = "test_timeseries_table";
    DeleteTimeseriesMetaRequest deleteTimeseriesMetaRequest = new DeleteTimeseriesMetaReque
st(tableName);
    deleteTimeseriesMetaRequest.setTimeseriesKeys(timeseriesKeyList);
    DeleteTimeseriesMetaResponse deleteTimeseriesMetaResponse = client.deleteTimeseriesMeta
(deleteTimeseriesMetaRequest);
    // Check whether the metadata of the time series is deleted.
    if (!deleteTimeseriesMetaResponse.isAllSuccess()) {
        for (DeleteTimeseriesMetaResponse.FailedRowResult failedRowResult : deleteTimeserie
sMetaResponse.getFailedRows()) {
            System.out.println(failedRowResult.getIndex());
            System.out.println(failedRowResult.getError());
        }
    }
}
```

2.14. Error handling

This topic describes the error handling methods in Tablestore SDK for Java.

Methods

Tablestore SDK for Java handles errors as exceptions. The operation is successful if no exceptions are returned. If an exception is returned, the operation fails.

> **Note** Batch operations such as BatchGetRow and BatchWriteRow are successful only when the system verifies that no exception is returned and that the status of each row is successful.

Exceptions

ClientException and TableStoreException are common exceptions you may encounter when you use Tablestore SDK for Java. Both are inherited from RuntimeException.

- ClientException: an internal SDK exception such as an invalid parameter value.
- TableStoreException: a server error parsed from a server error message. TableStoreException provides you with the following information:
  - getHttpStatus(): an HTTP status code such as 200 or 404.
  - getErrorCode(): an error type string returned by Tablestore.
  - getRequestId(): the UUID that identifies a request. If a problem persists, record this request ID and submit a ticket.

Retry

- Tablestore SDK for Java automatically retries operations in which an error occurs. In the default retry policy, the maximum retry period is 10s. Retries are performed for throttling errors and internal server errors related to write operations.
- You can also customize a retry policy by inheriting the RetryStrategy class, and configure the custom retry policy when you create an OTSClient object.

Tablestore SDK for Java provides the following retry policies:

- DefaultRetryStrategy: the default retry policy. Only read operations are retried. The retry interval increases exponentially from 10 ms and the maximum retry period is 10s.
- AlwaysRetryStrategy: All operations are retried. A maximum of three retry attempts can be made. The retry interval increases exponentially from 4 ms, and the maximum retry interval is 1s.

3.Go SDK

3.1. Preface

This topic describes how to install and use Tablestore SDK for Go.

- Tablestore is activated. For more information, see the Tablestore product page.
- An AccessKey ID and an AccessKey secret are created. For more information, see Obtain an AccessKey pair.

Download and install Tablestore SDK for Go

- For the download path of Tablestore SDK for Go, visit SDK package (including package, source code and examples).
- For more information about how to install Tablestore SDK for Go, see Installation.

Version

Latest version: 5.0.2

3.2. Installation

This topic describes how to install Tablestore SDK for Go.

Prerequisites

Go 1.4 or later is available.

Install Tablestore SDK for Go

Run the following command to install Tablestore SDK for Go:

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

3.3. Initialization

TableStoreClient is a client for Tablestore. TableStoreClient provides various methods that you can use to manage tables and perform read and write operations on a single row or multiple rows. Before you can use the TimeSeries model, you must initialize the TimeseriesClient.

Obtain an endpoint

An endpoint is a domain name that is used to access a Tablestore instance in a region. To query an endpoint, perform the following steps:

1. Log on to the Tablestore console.
2. Click the name of the instance to go to the **Instance Details** tab.

   On the Instance Details tab, you can view the endpoints of the instance in the Instance Access URL section.

> **Note** For more information about endpoints, see Endpoint.

Configure an AccessKey pair

Before you can connect to Tablestore, you must have an AccessKey pair that consists of an AccessKey ID and an AccessKey secret. The AccessKey pair is used to verify your identity. Tablestore supports the following types of AccessKey pairs:

- The AccessKey pair of an Alibaba Cloud account. To obtain the AccessKey pair of an Alibaba Cloud account, perform the following steps:
  i. On the Alibaba Cloud official website, register an Alibaba Cloud account.
  ii. Create an AccessKey pair in the RAM console.
- The AccessKey pair of a RAM user that is authorized to access Tablestore. To obtain the AccessKey pair of a RAM user, perform the following steps:
  i. Log on to the RAM console by using an Alibaba Cloud account. Then, create a RAM user or find an existing RAM user.
  ii. Grant the RAM user the permissions to access Tablestore.
  iii. After the RAM user is granted the permissions to access Tablestore, you can use the AccessKey pair of the RAM user to access Tablestore.
- Temporary access credentials that are obtained from Security Token Service (STS). The following process shows how temporary access credentials are obtained from STS:
  i. The application server uses RAM or STS to obtain access credentials that consist of a temporary AccessKey ID, an AccessKey secret, and a token, and then sends the access credentials to you.
  ii. You can use the access credentials to access Tablestore.

Initialize the TableStoreClient

After you obtain the AccessKey ID and AccessKey secret, you can initialize the TableStoreClient. The following sample code shows how to initialize the TableStoreClient.

- Operation

```
// Initialize a TableStoreClient instance.
// endPoint specifies the endpoint that is used to access the Tablestore instance. The en
dpoint must start with https:// or http://. Example: https://instance.cn-hangzhou.ots.ali
yun.com:80.
// accessKeyId specifies the AccessKey ID that is used to access the Tablestore instance.
To obtain an AccessKey ID, you can visit the official website of Alibaba Cloud or contact
an administrator.
// accessKeySecret specifies the AccessKey secret that is used to access the Tablestore i
nstance. To obtain an AccessKey secret, you can visit the official website of Alibaba Clo
ud or contact an administrator.
// instanceName specifies the name of the instance that you want to access. You can creat
e an instance in the Tablestore console or contact an administrator to obtain the name of
an existing instance.
func NewClient(endPoint, instanceName, accessKeyId, accessKeySecret string, options ...Cl
ientOption) *TableStoreClient
```

- Example

```
client = tablestore.NewClient("your_instance_endpoint",  "your_instance_name", "your_user
_id", "your_user_key")
```

Initialize the TimeseriesClient

After you obtain the AccessKey ID and AccessKey secret, you can initialize the TimeseriesClient. The following sample code shows how to initialize the TimeseriesClient.

- Operation

```
// Initialize a TimeseriesClient instance.
// endPoint specifies the endpoint that is used to access the Tablestore instance. The in
stance must start with https:// or http://. Example: https://instance.cn-hangzhou.ots.ali
yun.com:80.
// accessKeyId specifies the AccessKey ID that is used to access the Tablestore instance.
To obtain an AccessKey ID, you can visit the official website of Alibaba Cloud or contact
an administrator.
// accessKeySecret specifies the AccessKey secret that is used to access the Tablestore i
nstance. To obtain an AccessKey secret, you can visit the official website of Alibaba Clo
ud or contact an administrator.
// instanceName specifies the name of the instance that you want to access. You can creat
e an instance in the Tablestore console or contact an administrator to obtain the name of
an existing instance.
func NewTimeseriesClient(endPoint, instanceName, accessKeyId, accessKeySecret string , op
tions ...TimeseriesClientOption) *TimeseriesClient
```

- Example

```
timeseriesClient = tablestore.NewTimeseriesClient("your_instance_endpoint",  "your_instan
ce_name", "your_user_ak_id", "your_user_ak_key")
```

3.4. Table

3.4.1. Create a data table

This topic describes how to create a data table by calling the CreateTable operation. When you call the CreateTable operation, you must specify the schema information and configuration information of the data table when you call the CreateTable operation. You can set reserved read throughput and reserved write throughput for data tables in high-performance instances. You can create one or more index tables when you create a data table.

> **Note**
>
> - After a data table is created, it takes several seconds to load the data table. During this period, read and write operations performed on the table may fail. Perform operations on the data table after the data table is loaded.
> - You must specify the primary key when you create a data table. A primary key can contain one to four primary key columns. Each primary key column has a name and a data type.

Prerequisites

- An instance is created in the Tablestore console. For more information, see Create instances.

- A Tablestore client is initialized. For more information, see Initialization.

Operation

```
  // Description: Create a data table based on the specified table schema.
  // request is an instance of the CreateTableRequest class and contains TableMeta, TableOp
tion, and ReservedThroughput.
  // For more information, see the documentation of the TableMeta class.
  // After you create a data table, it takes several seconds to load the partitions in the
data table. You can perform operations on the data table only after the partitions are load
ed.
  // Response: CreateTableResponse
        CreateTable(request *CreateTableRequest) (*CreateTableResponse, error)
```

Parameters

Parameter	Description
TableMeta	The schema information of the data table. The schema information contains the following items: • TableName: the name of the data table. • PrimaryKey: the primary key of the data table. For more information, see Primary keys and attributes. **Note** Attribute columns are optional. Different rows in Tablestore can have different attribute columns. You can specify the names of attribute columns when you write data to the attribute columns. ◦ The primary key of a data table can contain one to four primary key columns. Primary key columns are sorted in the order in which they are added. For example, PRIMARY KEY (A, B, C) and PRIMARY KEY (A, C, B) have different primary key schemas. Tablestore sorts rows based on the values of all primary keys. ◦ The first primary key column is the partition key. Data that has the same partition key is stored in the same partition. Therefore, we recommend that you assign no more than 10 GB of data with the same partition key. Otherwise, a single partition may be too large to split. We recommend that you evenly distribute requests to read data from and write data to different partition keys for load balancing. • DefinedColumns: the predefined columns of the data table and the data types of the predefined columns. Primary key columns cannot be specified as predefined columns. You can use predefined columns as the index columns or attribute columns for index tables.
	The configuration information of the data table. For more information, see Data versions and TTL. The configurations contain the following items: • TimeToAlive: the period for which data in the table is retained. This

Parameter	Description
TableOption	period is the validity period of data. When the retention period exceeds the TimeToAlive value, Tablestore automatically deletes expired data. The minimum TimeToAlive value is 86400, which is equal to one day. A value of -1 specifies that data never expires. When you create a data table, you can set TimeToAlive to -1. This way, the data in the table never expires. After the table is created, you can call the UpdateTable operation to change the TimeToAlive value. Unit: seconds. **Note** If you want to create an index table for a data table, you must set TimeToAlive to -1. • MaxVersion: the maximum number of versions of data that can be retained for a single attribute column. When the number of versions of data in attribute columns exceeds the MaxVersion value, Tablestore automatically deletes earlier versions of data. When you create a data table, you can customize the maximum number of versions that can be retained for the data in an attribute column. After the data table is created, you can call the UpdateTable operation to modify the MaxVersion value for the data table. **Note** If you want to create an index table for a data table, you must set MaxVersion to 1. • DeviationCellVersionInSec: the max version offset, which is the maximum difference between the current system time and the timestamp of the written data. The difference between the version number and the time at which data is written must be within the value of DeviationCellVersionInSec. Otherwise, an error occurs when you write data. Unit: seconds. The valid version range of data in an attribute column is calculated based on the following formula: Valid version range = **[Data written time - Max version offset, Data written time + Max version offset)**. When you create a data table, Tablestore uses the default value of 86400 if you do not specify a max version offset. After the data table is created, you can call the UpdateTable operation to modify the max version offset value. Unit: seconds.

Parameter	Description
ReservedThroughput	The reserved read throughput and reserved write throughput of the data table. For data tables in capacity instances, the reserved read and write throughput can be set only to 0. Reserved throughput does not apply to data tables in these instances. The default value 0 specifies that all throughput is billed on a pay-as-you-go basis. Unit: capacity unit (CU). • If reserved read throughput or reserved write throughput is set to a value greater than 0 for a data table, Tablestore reserves related resources for the data table. After the data table is created, Tablestore charges reserved throughput resources. Additional throughput is billed on a pay-as-you-go basis. For more information, see Billing overview. • If reserved read throughput or reserved write throughput is set to 0 for a data table, Tablestore does not allocate or reserve resources for the data table.

Parameter	Description
IndexMeta	The schema information of the index table. The schema information contains the following items: • IndexName: the name of the index table. • PrimaryKey: the index columns of the index table. The index columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. • DefinedColumns: the attribute columns of the index table. The attribute columns are a combination of predefined columns of the data table. • IndexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ◦ If IndexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. When you use a global secondary index, Tablestore automatically synchronizes the data from the index columns and primary key columns of a data table to the columns of an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ◦ If IndexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. When you use the local secondary index feature, Tablestore automatically synchronizes the data from the index columns and primary key columns of a data table to the columns of an index table in synchronous mode. After data is written to the data table, you can query the data from the index table.

Examples

- Create a data table without creating an index table

  The following sample code provides an example on how to create a data table that has two primary key columns and reserved read and write throughput of (0, 0):

```
func CreateTableSample(client *tablestore.TableStoreClient, tableName string) {
    createTableRequest := new(tablestore.CreateTableRequest)
    // Create a schema for primary key columns, which includes the number, names, and typ
es of the primary key columns.
    // The first primary key column is named pk0 and requires an INTEGER value. The first
primary key column is also the partition key.
    // The second primary key column is named pk1 and requires an INTEGER value.
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = tableName
    tableMeta.AddPrimaryKeyColumn("pk0", tablestore.PrimaryKeyType_INTEGER)
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1
    tableOption.MaxVersion = 3
    reservedThroughput := new(tablestore.ReservedThroughput)
    reservedThroughput.Readcap = 0
    reservedThroughput.Writecap = 0
    createTableRequest.TableMeta = tableMeta
    createTableRequest.TableOption = tableOption
    createTableRequest.ReservedThroughput = reservedThroughput
    response, err := client.CreateTable(createTableRequest)
    if (err != nil) {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create table finished")
    }
}
```

For the detailed sample code, visit CreateTable@GitHub.

- Create a data table and an index table whose index type is global secondary index

```
func CreateTableWithGlobalIndexSample(client *tablestore.TableStoreClient, tableName stri
ng) {
    createTableRequest := new(tablestore.CreateTableRequest)
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = tableName
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableMeta.AddPrimaryKeyColumn("pk2", tablestore.PrimaryKeyType_INTEGER)
    tableMeta.AddDefinedColumn("definedcol1", tablestore.DefinedColumn_STRING)
    tableMeta.AddDefinedColumn("definedcol2", tablestore.DefinedColumn_INTEGER)
    indexMeta := new(tablestore.IndexMeta) // Create IndexMeta for the index table.
    indexMeta.AddPrimaryKeyColumn("definedcol1") // Add a primary key column to the index
table. Specify the definedcol1 column of the data table as the primary key column of the
index table.
    indexMeta.AddDefinedColumn("definedcol2") // Add an attribute column to the index tab
le. Specify the definedcol2 column of the data table as the attribute column of the index
table.
    indexMeta.IndexName = "indexSample"
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1 // Specify the validity period of data. A value of -1 sp
ecifies that the data never expires. Unit: seconds. You must set TimeToAlive to -1 for a
data table that has index tables.
    tableOption.MaxVersion = 1 // Specify the maximum number of versions that can be reta
ined for each column. A value of 1 specifies that only the latest version is retained for
each column. You must set MaxVersion to 1 for a data table that has index tables.
    reservedThroughput := new(tablestore.ReservedThroughput)
    createTableRequest.TableMeta = tableMeta
    createTableRequest.TableOption = tableOption
    createTableRequest.ReservedThroughput = reservedThroughput
    createTableRequest.AddIndexMeta(indexMeta)
    _, err := client.CreateTable(createTableRequest)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create table finished")
    }
}
```

- Create a data table and an index table whose index type is local secondary index

```
func CreateTableWithLocalIndexSample(client *tablestore.TableStoreClient, tableName strin
g) {
    createTableRequest := new(tablestore.CreateTableRequest)
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = tableName
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableMeta.AddPrimaryKeyColumn("pk2", tablestore.PrimaryKeyType_INTEGER)
    tableMeta.AddDefinedColumn("definedcol1", tablestore.DefinedColumn_STRING)
    tableMeta.AddDefinedColumn("definedcol2", tablestore.DefinedColumn_INTEGER)
    indexMeta := new(tablestore.IndexMeta) // Create IndexMeta for the index table.
    indexMeta.IndexType = IT_LOCAL_INDEX // Set the index type to IT_LOCAL_INDEX.
    indexMeta.AddPrimaryKeyColumn("pk1") // Add a primary key column to the index table.
The first primary key column of the index table must be the same as the first primary key
column of the data table.
    indexMeta.AddPrimaryKeyColumn("definedcol1") // Add a primary key column to the index
table. Specify the definedcol1 column of the data table as the primary key column of the
index table.
    indexMeta.AddDefinedColumn("definedcol2") // Add an attribute column to the index tab
le. Specify the definedcol2 column of the data table as the attribute column of the index
table.
    indexMeta.IndexName = "indexSample"
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1 // Specify the validity period of data. A value of -1 sp
ecifies that the data never expires. Unit: seconds. You must set TimeToAlive to -1 for a
data table that has index tables.
    tableOption.MaxVersion = 1 // Specify the maximum number of versions that can be reta
ined for each column. A value of 1 specifies that only the latest version is retained for
each column. You must set MaxVersion to 1 for a data table that has index tables.
    reservedThroughput := new(tablestore.ReservedThroughput)
    createTableRequest.TableMeta = tableMeta
    createTableRequest.TableOption = tableOption
    createTableRequest.ReservedThroughput = reservedThroughput
    createTableRequest.AddIndexMeta(indexMeta)
    _, err := client.CreateTable(createTableRequest)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create table finished")
    }
}
```

3.4.2. UpdateTable

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

Operations

```
// Update the configurations of tableoptions and reservedthroughput for a table.
UpdateTable(request *UpdateTableRequest) (*UpdateTableResponse, error)
```

Examples

The following code provides an example on how to update the maximum number of versions of a table to five:

```
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")
}
```

For the complete sample code, visit UpdateTable@GitHub.

3.4.3. List the names of tables

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

> **Note** For more information about the ListTable operation, see ListTable.

Prerequisites

- TableStoreClient is initialized. For more information, see Initialization.
- A table is created.

API operations

```
// List all table names. If the operation succeeds, names of all tables are returned.
ListTable() (*ListTableResponse, error)
```

Examples

The following code provides an example on how to query the names of all tables in an 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)
    }
}
```

For the complete sample code, visit ListTable@GitHub.

3.4.4. Predefined columns

This topic describes how to add predefined columns to or delete predefined columns from a table. You can set predefined columns to the indexed columns or indexed attribute columns when you create a secondary index.

Prerequisites

- TableStoreClient is initialized. For more information, see Initialization.
- A table is created.

Add predefined columns

When you create a secondary index for a base table, if the base table does not contain predefined columns or the existing predefined columns are not suitable for creating a secondary index, you can add predefined columns to the base table.

- Parameters

Parameter	Description
TableName	The name of the base table.
DefinedColumns	The name of the predefined columns and the data types of the predefined column values. The predefined columns cannot be primary key columns. You can set predefined columns to the indexed columns or indexed attribute columns for the secondary index. This parameter includes the following configurations: ◦ Name: the name of the predefined column. ◦ ColumnType: the data type of the predefined column value.

- Examples

  Add the following predefined columns to the base table named sampleTable: definedColumnName01 (type: STRING), definedColumnName02 (type: INTEGER), and definedColumnName03 (type: STRING).

```
func AddDefinedColumn() {
    client := new(tablestore.TableStoreClient)
    addDefinedColumnRequest := new(tablestore.AddDefinedColumnRequest)
    addDefinedColumnRequest.AddDefinedColumn("definedColumnName01",tablestore.DefinedColu
mn_STRING)
    addDefinedColumnRequest.AddDefinedColumn("definedColumnName02",tablestore.DefinedColu
mn_INTEGER)
    addDefinedColumnRequest.AddDefinedColumn("definedColumnName03",tablestore.DefinedColu
mn_STRING)
    addDefinedColumnRequest.TableName = "sampleTable"
    _, err := client.AddDefinedColumn(addDefinedColumnRequest)
    if (err != nil) {
        fmt.Println("Failed to Add DefinedColumn with error:", err)
    } else {
        fmt.Println("Add DefinedColumn finished")
    }
}
```

Delete predefined columns

You can delete predefined columns that you no longer use from a base table.

- Parameters

Parameter	Description
TableName	The name of the base table.
DefinedColumns	The name of the predefined column.

- Examples

  The following code provides an example on how to delete the predefined columns definedColumnName01 and definedColumnName02 from the base table sampleTable:

```
func DeleteDefinedColumn(){
    client := new(tablestore.TableStoreClient)
    deleteDefinedColumnRequest := new(tablestore.DeleteDefinedColumnRequest)
    deleteDefinedColumnRequest.DefinedColumns = []string{"definedColumnName01","definedCo
lumnName02"}
    _, err := client.DeleteDefinedColumn(deleteDefinedColumnRequest)
    if (err != nil) {
        fmt.Println("Failed to delete DefinedColumn with error:", err)
    } else {
        fmt.Println("Delete DefinedColumn finished")
    }
}
```

3.4.5. Query the description of a table

> **Note** For more information about the DescribeTable operation, see DescribeTable.

Prerequisites

- TableStoreClient is initialized. For more information, see Initialization.
- A table is created.

API operations

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

Parameters

Parameter	Description
TableName	The name of the table.

Examples

The following code provides an example on how to query the description of a table:

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

For the detailed sample code, visit DescribeTable@GitHub.

3.4.6. Delete tables

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

> **Note** For more information about the DeleteTable operation, see DeleteTable.

Prerequisites

- TableStoreClient is initialized. For more information, see Initialization.
- A table is created.
- The index tables and search indexes for the table are deleted.

Operations

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

Parameters

Parameter	Description
TableName	The name of the table.

Examples

The following code provides an example on how to delete a specified table:

```
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")
}
```

For the complete sample code, visit DeleteTable@GitHub.

3.4.7. Configure an auto-increment primary key column

This topic describes how to configure an auto-increment primary key column. You cannot set a partition key to an auto-increment column. If you write data to a table that contains an auto-increment primary key column, you do not need to specify specific values for the auto-increment primary key column because Tablestore generates values for the auto-increment primary key column. Values generated for the auto-increment primary key column are unique and consecutive within the partition that shares the same partition key value.

Prerequisites

The OTSClient instance is initialized. For more information, see Initialization.

Usage notes

1. When you create a table, you cannot set the partition key to an auto-increment primary key column.

   An auto-increment primary key column can only be an integer column. The generated value for an auto-increment primary key column is a 64-bit signed long integer.

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

   If you want to obtain the value of the auto-increment column after data is written to the table, you can set ReturnType to RT_PK.

   When you query data, you must specify the values of all primary key columns. To obtain a complete primary key value, you can set ReturnType to RT_PK in PutRow, UpdateRow, or BatchWriteRow.

Examples

When you use auto-increment primary key columns, you can call the CreateTable, PutRow, UpdateRow, or BatchWriteRow operation.

1. Create a table

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

```
import (
    "fmt"
    "github.com/aliyun/aliyun-tablestore-go-sdk/tablestore"
)
func CreateTableKeyAutoIncrementSample(client *tablestore.TableStoreClient) {
    createtableRequest := new(tablestore.CreateTableRequest)
    // Create a table that contains three primary key columns: pk1 of the STRING type,
pk2 of the INTEGER type, and pk3 of the BINARY type. pk2 is the auto-increment primary
key column.
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = "incrementsampletable"
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableMeta.AddPrimaryKeyColumnOption("pk2", tablestore.PrimaryKeyType_INTEGER, table
store.AUTO_INCREMENT)
    tableMeta.AddPrimaryKeyColumn("pk3", tablestore.PrimaryKeyType_BINARY)
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1
    tableOption.MaxVersion = 3
    reservedThroughput := new(tablestore.ReservedThroughput)
    reservedThroughput.Readcap = 0
    reservedThroughput.Writecap = 0
    createtableRequest.TableMeta = tableMeta
    createtableRequest.TableOption = tableOption
    createtableRequest.ReservedThroughput = reservedThroughput
    client.CreateTable(createtableRequest)
}
```

2. Write data

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

```
import (
    "fmt"
    "github.com/aliyun/aliyun-tablestore-go-sdk/tablestore"
)
func PutRowWithKeyAutoIncrementSample(client *tablestore.TableStoreClient) {
    fmt.Println("begin to put row")
    putRowRequest := new(tablestore.PutRowRequest)
    putRowChange := new(tablestore.PutRowChange)
    putRowChange.TableName = "incrementsampletable"
    // Specify primary key columns, which must be added in the order specified when the
table is created. Set pk2 to the auto-increment primary key column.
    putPk := new(tablestore.PrimaryKey)
    putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
    putPk.AddPrimaryKeyColumnWithAutoIncrement("pk2")
    putPk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    putRowChange.PrimaryKey = putPk
    putRowChange.AddColumn("col1", "col1data1")
    putRowChange.AddColumn("col2", int64(100))
    putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    putRowRequest.PutRowChange = putRowChange
    _, err := client.PutRow(putRowRequest)
    if err != nil {
        fmt.Println("put row failed with error:", err)
    } else {
        fmt.Println("put row finished")
    }
}
```

3.4.8. Configure conditional update

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

When you use the PutRow, UpdateRow, DeleteRow, or BatchWriteRow operation to update data, you can use conditional update to check row existence conditions and column-based conditions. The update is successful only when the conditions are met.

Conditional update includes row existence conditions and column-based conditions.

- Row existence conditions: include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.

  When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported.

- Column-based conditions: include SingleColumnCondition and CompositeColumnCondition, which are used to determine whether the conditions are met based on the values of one or more columns.
  - SingleColumnCondition supports the comparison between a constant and a column that can be a primary key column. SingleColumnCondition does not support the comparison between two columns or two constants.

- The inner node of CompositeColumnCondition is used to perform logical operations. Subconditions can be SingleColumnCondition or CompositeColumnCondition.

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

Parameters

Parameter	Description
RowExistenceExpectation	When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported. Row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST. In Tablestore, RowExistenceExpectation_IGNORE indicates IGNORE, RowExistenceExpectation_EXPECT_EXIST indicates EXPECT_EXIST, and RowExistenceExpectation_EXPECT_NOT_EXIST indicates EXPECT_NOT_EXIST. • IGNORE: No existence check is performed. • EXPECT_EXIST: The row is expected to exist. If the row exists, the condition is met. If the row does not exist, the condition is not met. • EXPECT_NOT_EXIST: Expect that the row does not exist. If the row does not exist, the condition is met. If the row exists, the condition is not met.
ColumnName	The column name.
ColumnValue	The comparison value of the column.
Comparator	The relational operator that is used to compare column values. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=). In Tablestore, CT_EQUAL indicates EQUAL (=), CT_NOT_EQUAL indicates NOT_EQUAL (!=), CT_GREATER_THAN indicates GREATER_THAN (>), CT_GREATER_EQUAL indicates GREATER_EQUAL (>=), CT_LESS_THAN indicates LESS_THAN (<), and CT_LESS_EQUAL indicates LESS_EQUAL (<=).
Operator	The logical operator that is used to combine multiple conditions. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. In Tablestore, LO_NOT indicates NOT, LO_AND indicates AND, and LO_OR indicates OR. Different logical operators require different number of subconditions. • When the logical operator is NOT, only one subcondition can be added. • When the logical operator is AND or OR, you must add at least two subconditions.

Parameter	Description
FilterIfMissing	Specifies whether to pass the conditional check when a column does not exist in a row. The type of this parameter value is Boolean. The default value is true, which indicates that if the column does not exist in a row, the conditional check is passed, and the row meets the update conditions. When FilterIfMissing is set to false, if the column does not exist in a row, the conditional check fails, and the row does not meet the update conditions.
LatestVersionsOnly	Specifies whether to use only the value of the latest version when the comparison value has multiple versions. The type of this parameter value is Boolean. The default value is true, which indicates that if the column has multiple versions of data, only the value of the latest version is used for comparison. When LatestVersionsOnly is set to false, the comparison values of all versions are used for comparison if the comparison value has multiple versions. In this case, if only the value of a version meets the condition, the conditional check is passed, and the row meets the update conditions.

Examples

- Example 1

  Update a row based on the specified primary key. If the specified row exists, the update succeeds. Otherwise, the update fails.

```
    updateRowRequest := new(tablestore.UpdateRowRequest)
    updateRowChange := new(tablestore.UpdateRowChange)
    updateRowChange.TableName = tableName
    updatePk := new(tablestore.PrimaryKey)
    updatePk.AddPrimaryKeyColumn("pk1", "pk1value1")
    updatePk.AddPrimaryKeyColumn("pk2", int64(2))
    updatePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    updateRowChange.PrimaryKey = updatePk
    updateRowChange.DeleteColumn("col1")            // Delete the col1 column.
    updateRowChange.PutColumn("col2", int64(77))    // Add the col2 column whose value is
77.
    updateRowChange.PutColumn("col4", "newcol3")    // Add the col4 column whose value is
"newcol3".
    // Expect that the specified row exists.
    updateRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_EXIST)
    updateRowRequest.UpdateRowChange = updateRowChange
    _, err := client.UpdateRow(updateRowRequest)
```

- Example 2

  Delete a row based on the specified primary key. If the specified row exists and the value of col2 is 3, the update succeeds. Otherwise, the update fails.

```
    deleteRowReq := new(tablestore.DeleteRowRequest)
    deleteRowReq.DeleteRowChange = new(tablestore.DeleteRowChange)
    deleteRowReq.DeleteRowChange.TableName = tableName
    deletePk := new(tablestore.PrimaryKey)
    deletePk.AddPrimaryKeyColumn("pk1", "pk1value1")
    deletePk.AddPrimaryKeyColumn("pk2", int64(2))
    deletePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    deleteRowReq.DeleteRowChange.PrimaryKey = deletePk
    // Expect that the specified row exists.
    deleteRowReq.DeleteRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_E
XIST)
    // Expect that the value in the col2 column is 3.
    clCondition1 := tablestore.NewSingleColumnCondition("col2", tablestore.CT_EQUAL, int6
4(3))
    deleteRowReq.DeleteRowChange.SetColumnCondition(clCondition1)
    _, err := client.DeleteRow(deleteRowReq)
```

3.4.9. Configure local transaction

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

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

You can use local transaction to specify that the operations on a partition key are atomic, which indicates that operations based on the specified partition key either succeed or fail. The isolation level of the local transaction is read committed.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

1. Use StartLocalTransaction to create a local transaction based on the specified partition key value and obtain the ID of the local transaction.
2. Read and write data within the local transaction.

   You can call the following operations on the local transaction: GetRow, PutRow, DeleteRow, UpdateRow, BatchWriteRow, and GetRange.
3. Use CommitTransaction to commit local transactions or use AbortTransaction to abort local transactions.

Limits

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

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

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

    If a transaction is not read or written for more than 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.
  - Up to 4 MB of data can be written to each transaction. The volume of data written to each transaction is calculated in the same way as a regular write request.
  - If you do not specify a version number for a cell, the Tablestore server assigns a version number to the cell in the usual way when the cell is written to the transaction (rather than when the transaction is committed).
  - If a BatchWriteRow request includes a transaction ID, all rows in the request can be written only to the table that matches the transaction ID.
  - When you use a transaction, the data within the range specified based on the corresponding partition key value is locked. If a request that is sent to write data within the local transaction does not contain the transaction ID, the request fails. Data within the transaction is unlocked when the transaction is committed or aborted, or when the transaction times out.
  - A transaction remains valid even if a read or write request with the transaction ID is rejected. You can resend the request in the same manner as a regular request or you can abort the transaction.

Parameters

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

Examples

1. Use StartLocalTransaction to create a local transaction based on the specified partition key value

and obtain the ID of the local transaction.

```
    // Specify a partition key for the local transaction. The partition key is the firs
t primary key column.
    transPk := new(tablestore.PrimaryKey)
    transPk.AddPrimaryKeyColumn("userid", userName)
    trans := &tablestore.StartLocalTransactionRequest{
        TableName: TableName,
        PrimaryKey: transPk,
    }
    response, err := client.StartLocalTransaction(trans)
    if err != nil {
        fmt.Println("failed to create transaction", err)
        return
    }
    // Obtain the local transaction ID.
    transId := response.TransactionId
```

2. Read and write data within the local transaction.

   Local transaction IDs are used to read and write data within the local transaction, which is the same as the normal read and write operations on data.

   - Write a row of data.

     ```
     putPk := new(tablestore.PrimaryKey)
     putPk.AddPrimaryKeyColumn("userid", userName)
     putPk.AddPrimaryKeyColumn("age", int64(18))
     putRowChange := &tablestore.PutRowChange{
         TableName: tableName,
         PrimaryKey: putPk,
     }
     putRowChange.AddColumn("col1", "col1data1")
     putRowChange.AddColumn("col2", int64(3))
     putRowChange.AddColumn("col3", []byte("test"))
     putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
     // Set the local transaction ID, which can be obtained by using StartLocalTransac
     tionResponse.TransactionId.
     putRowChange.TransactionId = transId
     putRowRequest := &tablestore.PutRowRequest{
         PutRowChange: putRowChange,
     }
     _, err = client.PutRow(putRowRequest)
     ```

   - Read the row of data.

```
getRowPk := new(tablestore.PrimaryKey)
getRowPk.AddPrimaryKeyColumn("userid", userName)
getRowPk.AddPrimaryKeyColumn("age", int64(18))
criteria := &tablestore.SingleRowQueryCriteria{
    PrimaryKey: getRowPk,
    TableName: tableName,
    // Read the latest version of data.
    MaxVersion: 1,
    // Set the local transaction ID.
    TransactionId: transId,
}
getRowRequest := &tablestore.GetRowRequest{
    SingleRowQueryCriteria: criteria,
}
getResp, err := client.GetRow(getRowRequest)
```

3. Commit or abort a local transaction.
   - Commit a local transaction for all modified data to take effect.

     ```
     request := &tablestore.CommitTransactionRequest{
         TransactionId: transId,
     }
     commitResponse, err := client.CommitTransaction(request)
     ```

   - Abort a local transaction. Modifications to the data within the local transaction are ignored.

     ```
     request := &tablestore.AbortTransactionRequest{
         TransactionId: transId,
     }
     abortResponse, err := client.AbortTransaction(request)
     ```

3.4.10. Configure atomic counter

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

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Limits

- You can implement atomic counters only on INTEGER columns.
- By default, if a column that is specified as an atomic counter does not exist, the value of the column is 0 before you write data. If a column that is specified as an atomic counter is not an INTEGER column, an OTSParameterInvalid error occurs.
- You can update an atomic counter by using a positive or negative number, but you must avoid an integer overflow. If an integer overflow occurs, an OTSParameterInvalid error is returned.
- By default, the value of an atomic counter is not returned in the response to an update row request. You can specify that the increased value of an atomic counter is returned.
- You cannot specify a column as an atomic counter and update the column in a single request. If

Attribute Column A is set to an atomic counter, you cannot perform other operations such as overwrite and delete operations on the attribute column A.

- You can perform multiple update operations on the same row by using a BatchWriteRow request. However, if you perform an atomic counter operation on a row, you can perform only one update operation on the row in a BatchWriteRow request.
- Only the value of the latest version of an atomic counter is increased. You cannot increase the value of a specified version of an atomic counter. After you update a row, a new version of data is inserted to the atomic counter in the row.

API operations

The API operations to implement the atomic counter feature are added to the updateRowChange class. The following table lists these operations.

Operation	Description
IncrementColumn(columnName string, value int64)	Increases or decreases a value by a number.
AppendIncrementColumnToReturn(name string)	Specifies the name of the column on which operations are performed to implement atomic counter. This API operation is configured to return the value of the column.
SetReturnIncrementValue()	Specifies the return type. The updated value in the column on which operations are performed to implement atomic counter must be returned.

Parameters

Parameter	Description
TableName	The name of the table.
ColumnName	The name of the column on which operations are performed to implement atomic counter. Columns whose valid values are integers are supported to implement the atomic counter feature.
Value	The value to update the columns on which operations are performed to implement atomic counter.

Examples

The following code provides an example on how to call the updateRowChange operation to update the columns whose valid values are integers and return the updated values when you write a row of data:

```
func UpdateRowWithIncrement(client *tablestore.TableStoreClient, tableName string) {
    fmt.Println("begin to update row")
    updateRowRequest = new(tablestore.UpdateRowRequest)
    updateRowChange = new(tablestore.UpdateRowChange)
    // Specify the name of the table.
    updateRowChange.TableName = tableName
    updatePk = new(tablestore.PrimaryKey)
    updatePk.AddPrimaryKeyColumn("pk1", "pk1increment")
    updatePk.AddPrimaryKeyColumn("pk2", int64(2))
    updatePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    updateRowChange.PrimaryKey = updatePk
    // Specify that the value the col2 column on which operations are performed to implemen
t atomic counter is increased by 30. Do not set timestamps.
    updateRowChange.IncrementColumn("col2", int64(30))
    // Return the values of the columns on which operations are performed to implement atom
ic counter.
    updateRowChange.SetReturnIncrementValue()
    updateRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    updateRowChange.AppendIncrementColumnToReturn("col2")
    updateRowRequest.UpdateRowChange = updateRowChange
    resp, err := client.UpdateRow(updateRowRequest)
    if err != nil {
        fmt.Println("update failed with error:", err)
        return
    } else {
        fmt.Println("update row finished")
        fmt.Println(resp)
        fmt.Println(len(resp.Columns))
        fmt.Println(resp.Columns[0].ColumnName)
        fmt.Println(resp.Columns[0].Value)
        fmt.Println(resp.Columns[0].Timestamp)
    }
}
```

3.4.11. Configure filter

Tablestore filters query results on the server side before returning the results. Only rows of data that match the filter conditions are returned.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

When you call the GetRow, BatchGetRow, or GetRange operation to query data, you can use a filter to return only the rows that meet the filter conditions.

The available filters are SingleColumnValueFilter, SingleColumnValueRegexFilter, and CompositeColumnValueFilter.

- SingleColumnValueFilter determines whether to filter a row based only on the values of a reference

column.

- SingleColumnValueRegexFilter uses regular expressions to match column values of the String type and extract matching substrings. Then, this filter converts the data type of the extracted substrings to String, Integer, or Double and filters the values after conversion.
- CompositeColumnValueFilter determines whether to filter a row by combining the filter conditions of values of multiple reference columns.

> **Note** For more information about filters, see Configure a filter in Function Introduction.

Limits

- The filter conditions support relational operators such as =, !=, >, >=, <, and <=, and logical operators such as NOT, AND, and OR. You can specify a maximum of 10 conditions.
- Reference columns in filters must be included in the read data. If the specified columns from which data is read do not include reference columns, the filter cannot query the values of reference columns.
- You can use filters by calling the GetRow, BatchGetRow, and GetRange operations, which does not change the native semantics or limited items of these operations.

  When you use GetRange, the number of rows scanned cannot exceed 5,000, or the data scanned cannot exceed 4 MB in size.

  If the scanned 5,000 rows or 4 MB of data does not match the filter conditions, the rows in the response are empty. However, NextStartPrimaryKey may not be empty. In this case, you must use NextStartPrimaryKey to continue reading the data until NextStartPrimaryKey is empty.

Parameters

Parameter	Description
ColumnName	The name of the reference column in the filter.
ColumnValue	The comparison value of the reference column in the filter.
ComparatorType	The relational operator in the filter. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=). In Tablestore, tablestore.CT_EQUAL indicates EQUAL (=), tablestore.CT_NOT_EQUAL indicates NOT_EQUAL (!=), tablestore.CT_GREATER_THAN indicates GREATER_THAN (>), tablestore.CT_GREATER_EQUAL indicates GREATER_EQUAL (>=), tablestore.CT_LESS_THAN indicates LESS_THAN (<), and tablestore.CT_LESS_EQUAL indicates LESS_EQUAL (<=).
LogicOperator	The logical operator in the filter. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. In Tablestore, tablestore. LO_NOT indicates NOT, tablestore.LO_AND indicates AND, and tablestore.LO_OR indicates OR.

Parameter	Description
FilterIfMissing	Specifies whether to return a row if the row does not contain the reference column. The type of this parameter value is Boolean. The default value is true, which indicates that if the reference column does not exist in a row, the row is returned. When FilterIfMissing is set to false, if the reference column does not exist in a row, the row is not returned.
LatestVersionOnly	Specifies whether to use only the value of the latest version for comparison when the reference column contains data of multiple versions. The type of this parameter value is Boolean. The default value is true, which indicates that if the reference column has multiple versions of data, only the value of the latest version is used for comparison. When LatestVersionsOnly is set to false, if a reference column has multiple versions of data, the value of all versions in the column is used for comparison. In this case, if only the value of one version meets the conditions, the row is returned.

Examples

- The following code provides an example on how to create SingleColumnValueFilter:

```
func GetRowWithFilter(client *tablestore.TableStoreClient, tableName string) {
    fmt.Println("begin to get row")
    pk := new(tablestore.PrimaryKey)
    pk.AddPrimaryKeyColumn("pk1", "pk1value1")
    pk.AddPrimaryKeyColumn("pk2", int64(2))
    pk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    // Specify that the rows are returned when the value of c1 is Zhejiang, and FilterIfM
issing is true.
    condition := tablestore.NewSingleColumnCondition("c1", tablestore.ComparatorType(tabl
estore.CT_EQUAL), "Zhejiang")
    condition.FilterIfMissing = true
    criteria := &tablestore.SingleRowQueryCriteria{
        TableName:     tableName,
        PrimaryKey:    pk,
        MaxVersion:    1,
        Filter:        condition,
    }
    getResp, err := client.GetRow(&tablestore.GetRowRequest{SingleRowQueryCriteria: crite
ria})
    if err != nil {
        fmt.Println("getrow failed with error:", err)
    } else {
        colMap := getResp.GetColumnMap()
        fmt.Println("length is ", len(colMap.Columns))
        fmt.Println("get row col0 result is ", getResp.Columns[0].ColumnName, getResp.Col
umns[0].Value)
    }
}
```

- The following code provides an example on how to create CompositeColumnValueFilter:

```
func GetRowWithCompositeColumnValueFilter(client *tablestore.TableStoreClient, tableName
string) {
    fmt.Println("begin to get row")
    pk := new(tablestore.PrimaryKey)
    pk.AddPrimaryKeyColumn("pk1", "pk1value1")
    pk.AddPrimaryKeyColumn("pk2", int64(2))
    pk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    // Specify that the rows are returned when the value of c1 is Zhejiang, and the value
of c2 is Hangzhou.
    filter := tablestore.NewCompositeColumnCondition(tablestore.LO_AND)
    filter1 := tablestore.NewSingleColumnCondition("c1", tablestore.CT_EQUAL, "Zhejiang")
    filter2 := tablestore.NewSingleColumnCondition("c2", tablestore.CT_EQUAL, "Hangzhou")
    filter.AddFilter(filter2)
    filter.AddFilter(filter1)
    criteria := &tablestore.SingleRowQueryCriteria{
        TableName:  tableName,
        PrimaryKey: pk,
        MaxVersion: 1,
        Filter:     filter,
    }
    getResp, err := client.GetRow(&tablestore.GetRowRequest{SingleRowQueryCriteria: crite
ria})
    if err != nil {
        fmt.Println("getrow failed with error:", err)
    } else {
        colMap := getResp.GetColumnMap()
        fmt.Println("length is ", len(colMap.Columns))
        fmt.Println("get row col0 result is ", getResp.Columns[0].ColumnName, getResp.Col
umns[0].Value)
    }
}
```

3.5. Single-row operations

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

PutRow

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

- Operations

```
    // @param PutRowRequest    Encapsulate the parameters required to call the PutRow ope
ration.
    // @return PutRowResponse
    PutRow(request *PutRowRequest) (*PutRowResponse, error)
```

- Parameters

Parameter	Description
TableName	The name of the table.
PrimaryKey	The primary key of the row. **Note** ◦ The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table. ◦ When the primary key column is an auto-increment column, you need only to set the value of the auto-increment column to the auto-increment primary key column. For more information, see Configure an auto-increment primary key column.
Columns	The attribute columns of the row. ◦ An attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType, which is optional), and timestamp (optional). ◦ You can set ColumnType to ColumnType.INTEGER, ColumnType.STRING, ColumnType.BINARY, ColumnType.BOOLEAN, or ColumnType.DOUBLE, which separately indicates INTEGER, STRING (a UTF-8 encoded string), BINARY, BOOLEAN, or DOUBLE. If you want to set the column value type to BINARY, you must set ColumnType to ColumnType.BINARY. If you want to use other types of column values, the setting of ColumnType is optional. ◦ A timestamp is the data version number. For more information, see Data versions and TTL. You can customize a data version number or use the data version number generated by Tablestore. If you do not specify this parameter, the data version number generated by the system is used. ■ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ■ When you choose to specify the version number, ensure that the version number is a 64-bit timestamp accurate to the millisecond within the valid version range.
Condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.

- Examples

  The following code provides an example on how to insert a row of data:

```
    putRowRequest := new(tablestore.PutRowRequest)
    putRowChange := new(tablestore.PutRowChange)
    putRowChange.TableName = tableName
    putPk := new(tablestore.PrimaryKey)
    putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
    putPk.AddPrimaryKeyColumn("pk2", int64(2))
    putPk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    putRowChange.PrimaryKey = putPk
    putRowChange.AddColumn("col1", "col1data1")
    putRowChange.AddColumn("col2", int64(3))
    putRowChange.AddColumn("col3", []byte("test"))
    putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    putRowRequest.PutRowChange = putRowChange
    _, err := client.PutRow(putRowRequest)
    if err != nil {
        fmt.Println("putrow failed with error:", err)
    } else {
        fmt.Println("putrow finished")
    }
```

  For the detailed sample code, visit PutRow@GitHub.

GetRow

You can call this operation to read a row of data.

The following results of the read request may be returned:

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

```
    // Return a row of data in the table.
    //
    // @param GetRowRequest             Encapsulate the parameters required to call the G
etRow operation.
    // @return  GetRowResponse          The content of the response to the GetRow operati
on.
    GetRow(request *GetRowRequest) (*GetRowResponse, error)
```

- Parameters

Parameter	Description
TableName	The name of the table.

Parameter	Description
PrimaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
ColumnsToGet	The names of the columns to read. You can specify the names of the primary key columns and the names of attribute columns. If you do not specify this parameter, all data in the row is returned. **Note** ◦ By default, Tablestore returns the data from all columns of the row when you query a row. You can set the ColumnsToGet parameter to read the data only in specified columns. If col0 and col1 are added to ColumnsToGet, only the values of the col0 and col1 columns are returned. ◦ When ColumnsToGet and Filter are used at the same time, the columns specified by ColumnsToGet are returned. Then, the returned columns are filtered.
MaxVersion	The maximum number of versions that can be read. **Note** You must specify at least one of MaxVersion and TimeRange. ◦ If you specify only MaxVersion, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only TimeRange, all data within a range or a version of data is returned. ◦ If you specify both MaxVersion and TimeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
TimeRange	Specifies a range of versions to read or a version of data to read. For more information, see TimeRange. **Note** You must specify at least one of MaxVersion and TimeRange. ◦ If you specify only MaxVersion, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only TimeRange, all data within a range or a version of data is returned. ◦ If you specify both MaxVersion and TimeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ To query data within a range, you must set Start and End. Start specifies the start timestamp. End indicates the end timestamp. The time range is a left-closed and right-open interval, which is [Start, End). ◦ To query data of a specific version, set Specific. Specific specifies a specific timestamp. You can set one of Specific and [Start, End). The timestamp used for the value of time_range ranges from 0 to INT64.MAX. Unit: milliseconds.
Filter	You can set filter conditions to filter the queried results on the server side. Only rows that meet the specified filter conditions are returned. For more information, see Configure filter. **Note** When ColumnsToGet and Filter are used at the same time, the columns specified by ColumnsToGet are returned. Then, the returned columns are filtered.

- Examples

  The following code provides an example on how to read a row of data:

```
getRowRequest := new(tablestore.GetRowRequest)
    criteria := new(tablestore.SingleRowQueryCriteria);
    putPk := new(tablestore.PrimaryKey)
    putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
    putPk.AddPrimaryKeyColumn("pk2", int64(2))
    putPk.AddPrimaryKeyColumn("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.Colu
mns[0].Value,)
    }
```

For the detailed sample code, visit GetRow@GitHub.

UpdateRow

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

> **Note** If the UpdateRow request contains only columns to delete, and the specified row does not exist, a new row is not added.

- Operations

```
    // Update a row of data in a table.
    // @param UpdateRowRequest      Encapsulate the parameters required to call the Updat
eRow operation.
    // @return UpdateRowResponse    The content of the response to the UpdateRow operatio
n.
    UpdateRow(request *UpdateRowRequest) (*UpdateRowResponse, error)
```

- Parameters

Parameter	Description
TableName	The name of the table.
PrimaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.

Parameter	Description
Columns	The attribute columns of the row.
Condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.

- Examples

  The following code provides an example on how to update the data of a specified row:

```
    updateRowRequest := new(tablestore.UpdateRowRequest)
    updateRowChange := new(tablestore.UpdateRowChange)
    updateRowChange.TableName = tableName
    updatePk := new(tablestore.PrimaryKey)
    updatePk.AddPrimaryKeyColumn("pk1", "pk1value1")
    updatePk.AddPrimaryKeyColumn("pk2", int64(2))
    updatePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    updateRowChange.PrimaryKey = updatePk
    updateRowChange.DeleteColumn("col1")
    updateRowChange.PutColumn("col2", int64(77))
    updateRowChange.PutColumn("col4", "newcol3")
    updateRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_EXIST)
    updateRowRequest.UpdateRowChange = updateRowChange
    _, err := client.UpdateRow(updateRowRequest)
    if err != nil {
        fmt.Println("update failed with error:", err)
    } else {
        fmt.Println("update row finished")
    }
```

  For the detailed sample code, visit UpdateRow@GitHub.

DeleteRow

You can call this operation to delete a row. If the specified row does not exist, no change is made to the table.

- Operations

```
    // Delete a row of data from a table.
    // @param DeleteRowRequest           Encapsulate the parameters required to call the
DeleteRow operation.
    // @return DeleteRowResponse         The content of the response to the DeleteRow ope
ration.
    DeleteRow(request *DeleteRowRequest) (*DeleteRowResponse, error)
```

- Parameters

Parameter	Description
TableName	The name of the table.

Parameter	Description
PrimaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
Condition	You can use conditional update to set row existence conditions or column-based conditions. For more information, see Configure conditional update.

- Examples

  The following code provides an example on how to delete a row from a table:

```
    deleteRowReq := new(tablestore.DeleteRowRequest)
    deleteRowReq.DeleteRowChange = new(tablestore.DeleteRowChange)
    deleteRowReq.DeleteRowChange.TableName = tableName
    deletePk := new(tablestore.PrimaryKey)
    deletePk.AddPrimaryKeyColumn("pk1", "pk1value1")
    deletePk.AddPrimaryKeyColumn("pk2", int64(2))
    deletePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    deleteRowReq.DeleteRowChange.PrimaryKey = deletePk
    deleteRowReq.DeleteRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_E
XIST)
    clCondition1 := tablestore.NewSingleColumnCondition("col2", tablestore.CT_EQUAL, int6
4(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")
    }
```

  For the detailed sample code, visit DeleteRow@GitHub.

3.6. Multi-row operations

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

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

BatchWriteRow

You can call this operation to write several rows of data from one or more tables in a request. The BatchWriteRow operation consists of multiple PutRow, UpdateRow, and DeleteRow operations. The process of constructing an operation is the same as that of calling the PutRow, UpdateRow, and DeleteRow operations. BatchWriteRow supports conditional update.

Each individual operation of BatchWriteRow is independently performed. The result for each individual operation is independently returned.

When multiple rows are written, part of rows may fail to be written. If the operation fails for part of rows, an exception may not occur, but information about the index and error messages of the failed rows is stored in BatchWriteRowResponse. When you call the BatchWriteRow operation, you must check the response to determine whether the status of each row is successful. We recommend that you check the response in case of failed operations.

When the server detects incorrect parameters during operations, BatchWriteRow may throw a parameter exception when none of the operations are completed for the request.

- API operations

```
// Add, delete, or update multiple rows of data in multiple tables.
//
// @param BatchWriteRowRequest             Encapsulate the parameters required to call th
e BatchWriteRow operation.
// @return  BatchWriteRowResponse          The content of the response to the BatchWriteR
ow operation.
BatchWriteRow(request *BatchWriteRowRequest) (*BatchWriteRowResponse,error)
```

- Parameters

  For more information, see Single-row operations.

- Examples

  The following code provides an example on how to write 100 rows of data:

```
batchWriteReq := &tablestore.BatchWriteRowRequest{}
for i := 0; i < 100; i++ {
    putRowChange := new(tablestore.PutRowChange)
    putRowChange.TableName = tableName
    putPk := new(tablestore.PrimaryKey)
    putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
    putPk.AddPrimaryKeyColumn("pk2", int64(i))
    putPk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    putRowChange.PrimaryKey = putPk
    putRowChange.AddColumn("col1", "fixvalue")
    putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    batchWriteReq.AddRowChange(putRowChange)
}
response, err := client.BatchWriteRow(batchWriteReq)
if err != nil {
    fmt.Println("batch request failed with:", response)
} else {
    fmt.Println("batch write row finished")
}
```

  For the detailed sample code, visit BatchWriteRow@GitHub.

BatchGetRow

You can call this operation to read several rows of data from one or more tables in a request. BatchGetRow consists of multiple GetRow operations. The process of constructing an operation is the same as that of calling the GetRow operation. BatchGetRow supports conditional update.

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

Each individual operation of BatchGetRow is independently performed. The result for each individual operation is independently returned.

When multiple rows are read, part of rows may fail to be read. If the operation fails for part of rows, an exception may not occur, but the error messages of the failed rows is stored in BatchGetRowResponse. When you call the BatchWriteRow operation, you must check the response to determine whether the status of each row is successful.

- API operations

```
// Return multiple rows of data from a table.
//
// @param BatchGetRowRequest            Encapsulate the parameters required to call the
BatchGetRow operation.
// @return  BatchGetRowResponse         The content of the response to the BatchGetRow o
peration.
BatchGetRow(request *BatchGetRowRequest) (*BatchGetRowResponse, error)
```

- Parameters

  For more information, see Single-row operations.

- Examples

  The following code provides an example on how to read 10 rows of data at a time:

```
batchGetReq := &tablestore.BatchGetRowRequest{}
mqCriteria := &tablestore.MultiRowQueryCriteria{}
for i := 0; i < 10; i++ {
    pkToGet := new(tablestore.PrimaryKey)
    pkToGet.AddPrimaryKeyColumn("pk1", "pk1value1")
    pkToGet.AddPrimaryKeyColumn("pk2", int64(i))
    pkToGet.AddPrimaryKeyColumn("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")
}
```

  For the detailed sample code, visit BatchGetRow@GitHub.

GetRange

You can call this operation to read data within the specified range based on the primary key values.

The GetRange operation allows you to read data in a forward or backward direction within a specified range. You can also specify the number of rows to read. If the range is large and the number of scanned rows or the volume of data exceeds the limit, the scan stops and the read rows and the next primary key are returned. If only part of the 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.

The GetRange operation may stop and return data in the following situations:

- The amount of data read reaches 4 MB.
- The number of rows read reaches 5,000.
- The number of rows returned reaches the limit.
- All reserved read throughput is consumed. You have no reserved read throughput to read the next row of data.

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

- API operations

```
// Query multiple rows of data within a specified range in a table.
//
// @param GetRangeRequest            Encapsulate the parameters required to call the GetR
ange operation.
// @return GetRangeResponse          The content of the response to the GetRange operatio
n.
GetRange(request *GetRangeRequest) (*GetRangeResponse,error)
```

- Parameters

Parameter	Description
TableName	The name of the table.

Parameter	Description
Direction	The direction to read data. ◦ If the value is FORWARD, the value of the start primary key must be smaller than that of the end primary key. The returned rows are sorted in ascending order based on the primary key value. ◦ If the value is BACKWARD, the value of the start primary key must be greater than that of the end primary key. The returned rows are sorted in descending order based on the primary key value. Example: A table contains two primary keys A and B where the value of A is smaller than that of B. If you set direction to forward, the rows whose primary key column values are greater than or equal to the value of A and smaller than the value of B are returned in ascending order. If you set direction to backward, the rows whose primary key column values are greater than the value of A and smaller than or equal to the value of B are returned in descending order.
StartPrimaryKey EndPrimaryKey	Read data between the start primary key column and the end primary key column. The start and end points of ranges are composed of valid primary keys or virtual points INF_MIN and INF_MAX. The number of columns for the virtual point must be the same as that of the valid primary key. INF_MIN indicates an infinitely small value of which all other values of other types are larger. INF_MAX indicates an infinitely large value of which all other values are smaller. ◦ StartPrimaryKey indicates the primary key column from which the read starts. If the specified row exists, the row is included in the response. ◦ EndPrimaryKey indicates the primary key column at which the read ends. The end primary key column is excluded from the response even if the row exists. Rows in tables are sorted based on the primary key column values in ascending order. The range for reading is a left-closed and right-open interval. When data is read in the forward direction, all rows whose primary key column values are greater than or equal to the start primary key column value and smaller than the end primary key column value are returned.
Limit	The maximum number of rows to return. This value must be greater than 0. An operation stops when the maximum number of rows is returned in a forward or backward direction, even if a part of rows within the specified range are not returned. You can use NextStartPrimaryKey in the response to record the position where the read ends in this operation. This way, the next read can start from this position.

Parameter	Description
ColumnsToGet	The set of columns to read. The column name can be the primary key column or attribute column. If you do not specify this parameter, all data in the row is returned. **Note** ◦ If you query a row of data, the system returns the data in all columns of the row. You can set the ColumnsToGet parameter to read the data only in specified columns. If col0 and col1 are added to ColumnsToGet, only the values of the col0 and col1 columns are returned. ◦ If a row is within the specified range to read based on the primary key value but does not contain the specified columns to return, the response excludes the row. ◦ When ColumnsToGet and Filter are used at the same time, the columns specified by ColumnsToGet are returned. Then, the returned columns are filtered.
MaxVersions	The maximum number of read versions. **Note** You must specify at least one of MaxVersion and TimeRange. ◦ If you specify only MaxVersion, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only TimeRange, all data within a range or a version of data is returned. ◦ If you specify both MaxVersion and TimeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
TimeRange	Reads data within a range of versions or a specific version of data. For more information, see TimeRange. **Note** You must specify at least one of MaxVersion and TimeRange. ◦ If you specify only MaxVersion, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only TimeRange, all data within a range or a version of data is returned. ◦ If you specify both MaxVersion and TimeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ To query data within a range, you must set Start and End. Start specifies the start timestamp. End indicates the end timestamp. The time range is a left-closed and right-open interval, which is [Start, End). ◦ To query data of a specific version, set Specific. Specific specifies a specific timestamp. You can set one of Specific and [Start, End). Valid values: [0, INT64.MAX). Unit: milliseconds.
Filter	Filters the read results on the server side and returns only the rows of data that meet the conditions in the filter. For more information, see Configure filter. **Note** When ColumnsToGet and Filter are used at the same time, the columns specified by ColumnsToGet are returned. Then, the returned columns are filtered.
NextStartPrimaryKey	The start primary key of the next read. The value of NextStartPrimaryKey can be used to determine whether all data is read. ◦ If the value of NextStartPrimaryKey is not empty in the response, the NextStartPrimaryKey value can be used as the start primary key for the next GetRange operation. ◦ If the value of NextStartPrimaryKey is empty in the response, all data within the range is returned.

- Examples

  The following code provides an example on how to read data within a specified range:

```
getRangeRequest := &tablestore.GetRangeRequest{}
    rangeRowQueryCriteria := &tablestore.RangeRowQueryCriteria{}
    rangeRowQueryCriteria.TableName = tableName
    startPK := new(tablestore.PrimaryKey)
    startPK.AddPrimaryKeyColumnWithMinValue("pk1")
    startPK.AddPrimaryKeyColumnWithMinValue("pk2")
    startPK.AddPrimaryKeyColumnWithMinValue("pk3")
    endPK := new(tablestore.PrimaryKey)
    endPK.AddPrimaryKeyColumnWithMaxValue("pk1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk2")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk3")
    rangeRowQueryCriteria.StartPrimaryKey = startPK
    rangeRowQueryCriteria.EndPrimaryKey = endPK
    rangeRowQueryCriteria.Direction = tablestore.FORWARD
    rangeRowQueryCriteria.MaxVersion = 1
    rangeRowQueryCriteria.Limit = 10
    getRangeRequest.RangeRowQueryCriteria = rangeRowQueryCriteria
    getRangeResp, err := client.GetRange(getRangeRequest)
    fmt.Println("get range result is " ,getRangeResp)
    for {
        if err != nil {
            fmt.Println("get range failed with error:", err)
        }
        for _, row := range getRangeResp.Rows {
            fmt.Println("range get row with key", row.PrimaryKey.PrimaryKeys[0].Value, ro
w.PrimaryKey.PrimaryKeys[1].Value, row.PrimaryKey.PrimaryKeys[2].Value)
        }
        if getRangeResp.NextStartPrimaryKey == nil {
            break
        } else {
            fmt.Println("next pk is :", getRangeResp.NextStartPrimaryKey.PrimaryKeys[0].V
alue, getRangeResp.NextStartPrimaryKey.PrimaryKeys[1].Value, getRangeResp.NextStartPrimar
yKey.PrimaryKeys[2].Value)
            getRangeRequest.RangeRowQueryCriteria.StartPrimaryKey = getRangeResp.NextStar
tPrimaryKey
            getRangeResp, err = client.GetRange(getRangeRequest)
        }
        fmt.Println("continue to query rows")
    }
    fmt.Println("range get row finished")
```

For the detailed sample code, visit GetRange@GitHub.

3.7. Search Index

3.7.1. Create search indexes

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created for which the value of the TimeToLive parameter is set to -1 and the value of the MaxVersions parameter is set to 1.

Parameters

When you create a search index, you must configure the TableName, IndexName, and IndexSchema parameters. You must also configure the FieldSchemas, IndexSetting, and IndexSort parameters in IndexSchema. The following table describes the parameters.

Parameter	Description
TableName	The name of the data table.
IndexName	The name of the search index.
FieldSchemas	The list of field schemas. You can configure the following parameters for each field schema: • FieldName: This parameter is required and specifies the name of the field in the search index. The value is used as the column name. Type: String. A field in a search index can be a primary key column or an attribute column. • FieldType: This parameter is required and specifies the type of the field. Use tablestore.FieldType_XXX to set the type. For more information, see Data types of column values. • Array: This parameter is optional and specifies whether the value is an array. Type: Boolean. If you set this parameter to true, the column stores data as an array. Data written to the column must be a JSON array. Example: ["a","b","c"]. Nested values are an array. If you set FieldType to Nested, skip this parameter. • Index: This parameter is optional and specifies whether to enable indexing for the column. Type: Boolean. Default value: true. A value of true indicates that Tablestore indexes the column with an inverted indexing schema or a spatio-temporal indexing schema. A value of false indicates that Tablestore does not enable indexing for the column. • Analyzer: This parameter is optional and specifies the type of the analyzer that you want to use. If FieldType is set to Text, you can configure this parameter. Otherwise, the default analyzer type single-word tokenization is used. For more information about tokenization, see Tokenization. • EnableSortAndAgg: This parameter is optional and specifies whether to enable sorting and aggregation. Type: Boolean. Sorting can be enabled only for fields for which EnableSortAndAgg is set to true. For more information about sorting, see Sorting and pagination. Notice • Store: This parameter is optional and specifies whether to store the value of the field in the search index. Type: Boolean. If you set Store to true, you can read the value of the field from the search index without querying the data table. This improves query performance.

Parameter	Description
IndexSetting	The settings of the search index, including RoutingFields. RoutingFields: This parameter is optional and specifies custom routing fields. You can specify some primary key columns as routing fields. Tablestore distributes data that is written to a search index across different partitions based on the specified routing fields. The data whose routing field values are the same is distributed to the same partition.
IndexSort	The presorting settings for the search index, including Sorters. If no value is specified for the IndexSort parameter, field values are sorted by primary key. **Note** You can skip the presorting settings for the search indexes that contain the Nested field type. Sorters: This parameter is required and specifies the presorting method for the search index. PrimaryKeySort and FieldSort are supported. For more information about sorting, see Sorting and pagination. • PrimaryKeySort: Data is sorted by primary key. You can configure the following parameter for PrimaryKeySort: Order: the sort order. Data can be sorted in ascending or descending order. By default, data is sorted in ascending order. • FieldSort: Data is sorted by field value. You can configure the following parameters for FieldSort: Only fields for which both indexing and sorting and aggregation are enabled can be presorted. ◦ FieldName: the name of the field that is used to sort data. ◦ Order: the sort order. Data can be sorted in ascending or descending order. By default, data is sorted in ascending order. ◦ Mode: the sorting method that is used when the field contains multiple values.

Examples

- Create a search index

  The following sample code shows how to create a search index. The search index contains the col1 and col2 columns. The type of the col1 column is Keyword, and the type of the col2 column is Long.

```
func CreateSearchIndex(client *tablestore.TableStoreClient, tableName string, indexName s
tring) {
    request := &tablestore.CreateSearchIndexRequest{}
    request.TableName = "exampletable" // Specify the name of the data table.
    request.IndexName = "examplesearchindex" // Specify the name of the search index that
you want to create.
    schemas := []*tablestore.FieldSchema{}
    field1 := &tablestore.FieldSchema{
        FieldName: proto.String("Col_Keyword"), // Specify the field name by calling the
proto.String method. This method is used to request a string pointer.
        FieldType: tablestore.FieldType_KEYWORD, // Specify the field type.
        Index:     proto.Bool(true), // Enable indexing.
        EnableSortAndAgg: proto.Bool(true), // Enable sorting and aggregation.
    }
    field2 := &tablestore.FieldSchema{
        FieldName: proto.String("Col_Long"),
        FieldType: tablestore.FieldType_LONG,
        Index:     proto.Bool(true),
        EnableSortAndAgg: proto.Bool(true),
    }
    schemas = append(schemas, field1, field2)
    request.IndexSchema = &tablestore.IndexSchema{
        FieldSchemas: schemas, // Specify the fields that are included in the search inde
x.
    }
    resp, err := client.CreateSearchIndex(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)
}
```

- Create a search index with IndexSort specified

  The following sample code shows how to create a search index with IndexSort specified. The search index contains the col1 and col2 columns. The type of the col1 column is Keyword, and the type of the col2 column is Long.

```
func createSearchIndex_withIndexSort(client *tablestore.TableStoreClient){
    request := &tablestore.CreateSearchIndexRequest{}
    request.TableName = "exampletable" // Specify the name of the data table.
    request.IndexName = "examplesearchindex" // Specify the name of the search index that
you want to create.
    schemas := []*tablestore.FieldSchema{}
    field1 := &tablestore.FieldSchema{
        FieldName: proto.String("col1"), // Set the field name by calling the proto.Strin
g method. This method is used to request a string pointer.
        FieldType: tablestore.FieldType_KEYWORD, // Specify the field type.
        Index:     proto.Bool(true), // Enable indexing.
        EnableSortAndAgg: proto.Bool(true), // Enable sorting and aggregation.
    }
    field2 := &tablestore.FieldSchema{
        FieldName: proto.String("col2"),
        FieldType: tablestore.FieldType_LONG,
        Index:     proto.Bool(true),
        EnableSortAndAgg: proto.Bool(true),
    }
    schemas = append(schemas, field1, field2)
    request.IndexSchema = &tablestore.IndexSchema{
        FieldSchemas: schemas, // Specify the fields that are included in the search inde
x.
        IndexSort: &search.Sort{ // Specify the presorting order. The values in the col2
column are sorted in ascending order, and the values in the col1 column are sorted in des
cending order.
            Sorters: []search.Sorter{
                &search.FieldSort{
                    FieldName: "col2",
                    Order:     search.SortOrder_ASC.Enum(),
                },
                &search.FieldSort{
                    FieldName: "col1",
                    Order:     search.SortOrder_DESC.Enum(),
                },
            },
        },
    }
    resp, err := client.CreateSearchIndex(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)
}
```

3.7.2. List search indexes

You can call the ListSearchIndex operation to obtain indexes created in the current instance or associated with a table.

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

3.7.3. 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.

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.

Examples

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

3.7.4. Delete search indexes

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.

Examples

```
func DeleteSearchIndex(client *tablestore.TableStoreClient, tableName string, indexName str
ing) {
    request := &tablestore.DeleteSearchIndexRequest{}
    request.TableName = tableName // Set the name of the table.
    request.IndexName = indexName // Set the name of the search index.
    resp, err := client.DeleteSearchIndex(request) // Call a client to delete the search in
dex.
    if err != nil {
        fmt.Println("error: ", err)
        return
    }
    fmt.Println("DeleteSearchIndex finished, requestId: ", resp.ResponseInfo.RequestId)
}
```

3.7.5. Term query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
Query	The query type. To use term query, set this parameter to TermQuery.
FieldName	The name of the field used to match the query conditions.
Term	The keyword used for the query to match the column values. The keyword is not tokenized but used as an exact word to match the column values. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that can match the keyword. For example, if the value of a TEXT field in a row is "tablestore is cool", the value can be tokenized into "tablestore", "is", and "cool". The row can meet query conditions when you set "tablestore", "is", or "cool" to the keyword to match the values of the TEXT field.

Parameter	Description
ColumnsToGet	Specifies whether to return all columns of each matched row. You can configure ReturnAll and Columns for this parameter. By default, ReturnAll is set to false, which indicates that not all columns are returned. If ReturnAll is set to false, you can configure Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.
GetTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword exactly matches "hangzhou".
 */
func TermQuery(client *tablestore.TableStoreClient, 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 used to match the query conditions.
    query.Text = "hangzhou" // Set the keyword used for the query to match the column value
s.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchRequest.SetSearchQuery(searchQuery)
    // Set ReturnAll to true to return all columns.
    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) // Display the total number of m
atched 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))
    }
}
```

3.7.6. Terms query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
Query	The query type. To use terms query, set this parameter to TermsQuery.
FieldName	The name of the field used to match the query conditions.
Terms	The keywords used in the query to match the column values. A row meets the query conditions when the value of the specified column exactly matches at least one keyword.
Limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
ColumnsToGet	Specifies whether to return all columns of each matched row. You can configure ReturnAll and Columns for this parameter. By default, ReturnAll is set to false, which indicates that not all columns are returned. If ReturnAll is set to false, you can configure Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword exactly matches "hangzhou" or "
tablestore".
 */
func TermsQuery(client *tablestore.TableStoreClient, 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 field used to match the query conditions.
    terms := make([]interface{}, 0)
    terms = append(terms, "hangzhou")
    terms = append(terms, "tablestore")
    query.Terms = terms // Set the keywords used in the query to match the column values.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetLimit(100)
    searchRequest.SetSearchQuery(searchQuery)
    // Set ReturnAll to true to return all columns.
    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))
    }
}
```

3.7.7. Match all query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
Query	The query type. To use match all query, set this parameter to MatchAllQuery.
Limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
GetTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
ColumnsToGet	Specifies whether to return all columns of each matched row. By default, ReturnAll is set to false, which indicates that not all columns are returned. If ReturnAll is set to false, you can configure Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/**
 * Use match all query to query the total number of rows in a table.
 */
func MatchAllQuery(client *tablestore.TableStoreClient, 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, which indicates that no data is returned.
    searchRequest.SetSearchQuery(searchQuery)
    searchResponse, err := client.Search(searchRequest)
    if err != nil { // Check for exceptions.
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess)
    fmt.Println("TotalCount: ", searchResponse.TotalCount) // Display the total number of r
ows in the table.
}
```

3.7.8. Match query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
Query	The query type. To use match query, set this parameter to MatchQuery.
FieldName	The name of the column used to match the query conditions. Match query applies to TEXT columns.

Parameter	Description
Text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, you can specify a TEXT column for a query, and set the analyzer to perform single-word tokenization. If you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
Operator	The logical operator. By default, OR is used as the logical operator, which indicates that a row matches the query conditions when one of the tokenized keywords is matched. If you set operator to AND, a row meets the query condition only when all tokenized keywords are matched.
MinimumShouldMatch	The minimum number of matched tokenized keywords contained in a column value. A row is returned only when the value of the queried column in the row contains at least the minimum number of matched tokenized keywords. **Note** MinimumShouldMatch must be used together with the OR logical operator.
Offset	The position from which the current query starts.
Limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
GetTotalCount	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.

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

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword matches "hangzhou". Tablestore
returns the number of rows that match the phrase and parts of the matched rows in this quer
y.
 */
func MatchQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.MatchQuery{} // Set the query type to MatchQuery.
    settingquery.FieldName = "Col_Keyword" // Set the column used to match the query condit
ions.
    query.Text = "hangzhou" // Set the keyword used for the query to match the column value
s.
    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 { // Check for exceptions.
        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 mat
ched rows.
    fmt.Println("RowCount: ", len(searchResponse.Rows)) // Check the total number of return
ed rows.
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody)) // If ColumnsToGet is not specified, only pr
imary key columns that match the query conditions are returned.
    }
    // Set columnsToGet to true to return all columns.
```

```
    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 mat
ched 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))
    }
}
```

3.7.9. Phrase query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
Query	The query type. To use phrase query, set this parameter to MatchPhraseQuery.
FieldName	The name of the column used to match the query conditions. Match query applies to TEXT columns.

Parameter	Description
Text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore" and "this is a table". Query results such as "this table is ..." or "is this a table" cannot be obtained.
Offset	The position from which the current query starts.
Limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
GetTotalCount	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
ColumnsToGet	Specifies whether to return all columns of each matched rows. You can configure ReturnAll and Columns for this parameter. By default, ReturnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Text matches the whole phase "hangzhou
shanghai" in order. Tablestore returns the number of rows that match the phrase and parts o
f the matched rows in this query.
 */
func MatchPhraseQuery(client *tablestore.TableStoreClient, tableName string, indexName stri
ng) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.MatchPhraseQuery{} // Set the query type to MatchPhraseQuery.
    query.FieldName = "Col_Text"  // Set the column that you want to match.
    query.Text = "hangzhou shanghai"  // Set the keyword that you want to match.
    searchQuery := search.NewSearchQuery()
```

```
    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 mat
ched 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))
    }
    // Return all columns.
    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 mat
ched 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))
    }
}
```

3.7.10. Prefix query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword contains the prefix that exactl
y matches "hangzhou".
 */
func PrefixQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.PrefixQuery{} // Set the query type to PrefixQuery.
    query.FieldName = "Col_Keyword" // Set the field used to match the query conditions.
    query.Prefix = "hangzhou" // Specify the prefix.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchRequest.SetSearchQuery(searchQuery)
    // Set ReturnAll to true to return all columns.
    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) // Display the total number of m
atched 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))
    }
}
```

3.7.11. Range query

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.

Parameter	Description
IndexName	The name of the search index.
Query	The query type. To use range query, set this parameter to RangeQuery.
FieldName	The name of the field used to match the query conditions.
From	The value from which the query starts. You can use GT or GTE to specify a range. GT indicates that the values within the range are greater than the specified value. GTE indicates that the values within the range are greater than or equal to the specified value.
To	The value with which the query ends. You can use LT or LTE to specify a range. LT indicates that the values within the range are smaller than the specified value. LTE indicates that the values within the range are smaller than or equal to the specified value.
IncludeLower	Specifies whether to include the value of the From parameter in the response. The value type is Boolean.
IncludeUpper	Specifies whether to include the value of the To parameter in the response. The value type is Boolean.
Sort	The sorting method. The returned rows are sorted based on the specified method. For more information, see Sorting and pagination.
ColumnsToGet	Specifies whether to return all columns of each matched row. You can configure ReturnAll and Columns for this parameter. By default, ReturnAll is set to false, which indicates that not all columns are returned. If ReturnAll is set to false, you can configure Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/**
 * Search the table for rows where the value in Col_Long is greater than 3. Tablestore sort
s the matched rows based on the values of 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 to RangeQuery.
    rangeQuery.FieldName = "Col_Long" // Set the field used to match the query conditions.
    rangeQuery.GT(3) // Specify the range of the field value, which is greater than 3.
    searchQuery.SetQuery(rangeQuery)
    // Sort the query results based on the values of 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))
    }
}
```

3.7.12. Wildcard query

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

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the data table.
IndexName	The name of the search index.
query	The type of the query. Set the query type to WildcardQuery.
FieldName	The name of the column.
Value	The string that contains wildcards. The string cannot exceed 32 characters in length.
ColumnsToGet	Specifies whether to return all columns of each matched row. By default, the value of ReturnAll is false, which specifies that not all columns are returned. If ReturnAll is set to false, you can use ColumnsToGet to specify the columns that you want to return. If you do not specify the columns that you want to return, only the primary key columns are returned. If you set the ReturnAll parameter to true, all columns are returned.

Examples

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

 */
func WildcardQuery(client *tablestore.TableStoreClient, 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)
    // Specify that all columns are returned.
    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 matched
rows are returned.
    fmt.Println("TotalCount: ", searchResponse.TotalCount) // Specify that the total number
of matched rows instead of the number of returned rows is displayed.
    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))
    }
}
```

3.7.13. Geo query

Geo query includes the following query types: geo-distance query (GeoDistanceQuery), geo-bounding box query (GeoBoundingBoxQuery), and geo-polygon query (GeoPolygonQuery).

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Geo-distance query

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

- Parameters

Parameter	Description
FieldName	The column name. Type: Geopoint.
CenterPoint	The coordinate pair of the central point. The coordinate pair consists of latitude and longitude values. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of the longitude: [-180,180]. Example: "35.8,-45.91".
DistanceInMeter	The radius of the circle centered on the specified location. Type: Double. Unit: meters.
Query	The query statement for the search index. Sets the query type to GeoDistanceQuery.
TableName	The name of the table.
IndexName	The name of the search index.

- Examples

  The following code provides an example on how to 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.TableStoreClient, tableName string, indexName st
ring) {
    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"  // Specify the coordinate pair for a central point.
    query.DistanceInMeter = 10000.0 // Set the distance from the central point to a value
greater than or equal to 10,000. Unit: meters.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    // Return all columns.
    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) // Display 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))
    }
}
```

Geo-bounding box query

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

- Parameters

Parameter	Description
FieldName	The column name. Type: Geopoint.
TopLeft	The coordinate pair of the upper-left corner of the rectangular geographic area.

Parameter	Description
BottomRight	The coordinate pair of the lower-right corner of the rectangular geographic area. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of the longitude: [-180,180]. Example: "35.8,-45.91".
Query	The query statement for the search index. Sets the query type to GeoBoundingBoxQuery.
TableName	The name of the table.
IndexName	The name of the search index.

- Examples

  The data type of Col_GeoPoint is GeoPoint. The coordinate pair of the upper-left corner is "10,0". The coordinate pair of the lower-right corner is "0,10". The following code provides an example on how to obtain the rows where the value of Col_GeoPoint falls within the rectangular geographic area:

```
func GeoBoundingBoxQuery(client *tablestore.TableStoreClient, 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 column name.
    query.TopLeft = "10,0"  // Specify the coordinate pair for the upper-left corner of t
he rectangular geographic area.
    query.BottomRight = "0,10" // Specify coordinate pair for the lower-right corner of t
he rectangular geographic area.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    // Return all columns.
    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) // Display 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))
    }
}
```

Geo-polygon query

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

- Parameters

Parameter	Description
FieldName	The column name. Type: Geopoint.
Points	The coordinate pairs of the points that define a polygon area. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of the longitude: [-180,180]. Example: "35.8,-45.91".

Parameter	Description
Query	The query statement for the search index. Sets the query type to GeoPolygonQuery.
TableName	The name of the table.
IndexName	The name of the search index.

- Examples

  The following code provides an example on how to query the table for rows where the value of Col_GeoPoint falls within a specified geographic polygon area:

```
func GeoPolygonQuery(client *tablestore.TableStoreClient, tableName string, indexName str
ing) {
    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 coordinate pairs for vertices o
f a geographic polygon area.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    // Return all columns.
    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) // Display 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))
    }
}
```

3.7.14. Boolean query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the data table.
IndexName	The name of the search index.
MustQueries	The list of subqueries that the query results must match. This parameter is equivalent to the AND operator.
MustNotQueries	The list of subqueries that the query results must not match. This parameter is equivalent to the NOT operator.
FilterQueries	The list of subqueries. Only rows that match all subfilters are returned. filter is similar to query except that filter does not calculate the relevance score based on the number of subfilters that the row matches.
ShouldQueries	The list of subqueries that the query results can or cannot match. This parameter is equivalent to the OR operator. Only rows that meet the minimum number of subquery conditions specified by ShouldQueries are returned. A higher overall relevance score indicates that more subquery conditions specified by ShouldQueries are met.
MinimumShouldMatch	The minimum number of subquery conditions specified by ShouldQueries that the rows must meet. If no other subquery conditions except the subquery conditions that are specified by ShouldQueries are specified, the default value of the MinimumShouldMatch parameter is 1. If other subquery conditions, such as subquery conditions specified by MustQueries, MustNotQueries, and FilterQueries, are specified, the default value of the MinimumShouldMatch parameter is 0.

Example

The following sample code provides an example on how to perform a Boolean query based on a combination of subqueries:

```
/**
 * Perform a Boolean query based on a combination of subqueries.
 */
```

```
func BoolQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    /**
     * Condition 1: Perform a range query to query the rows where the Col_Long column value is greater than 3.
     */
    rangeQuery := &search.RangeQuery{}
    rangeQuery.FieldName = "Col_Long"
    rangeQuery.GT(3)
    /**
     * Condition 2: Perform a match query to query the rows where the Col_Keyword column value matches hangzhou.
     */
    matchQuery := &search.MatchQuery{}
    matchQuery.FieldName = "Col_Keyword"
    matchQuery.Text = "hangzhou"
    {
        /**
         * Construct a Boolean query where the rows must meet both Condition 1 and Condition 2.
         */
        boolQuery := &search.BoolQuery{
            MustQueries: []search.Query{
                rangeQuery,
                matchQuery,
            },
        }
        searchQuery := search.NewSearchQuery()
        searchQuery.SetQuery(boolQuery)
        searchRequest.SetSearchQuery(searchQuery)
        searchResponse, err := client.Search(searchRequest)
        if err != nil {
            fmt.Printf("%#v", err)
            return
        }
        fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // Check whether the returned results are complete.
        fmt.Println("TotalCount: ", searchResponse.TotalCount) // Specify that all rows that match the specified conditions are displayed. The number of rows to return is not displayed.
        fmt.Println("RowCount: ", len(searchResponse.Rows))
    }
    {
        /**
         * Construct a Boolean query where the rows meet at least one of Condition 1 and Condition 2.
         */
        boolQuery := &search.BoolQuery{
            ShouldQueries: []search.Query{
                rangeQuery,
                matchQuery,
            },
            MinimumShouldMatch: proto.Int32(1)
```

```
            MinimumShouldMatch: proto.Int32(1),
        }
        searchQuery := search.NewSearchQuery()
        searchQuery.SetQuery(boolQuery)
        searchRequest.SetSearchQuery(searchQuery)
        searchResponse, err := client.Search(searchRequest)
        if err != nil {
            fmt.Printf("%#v", err)
            return
        }
        fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // Check whether the ret
urned results are complete.
        fmt.Println("TotalCount: ", searchResponse.TotalCount) // Specify that all rows tha
t match the specified conditions are displayed. The number of rows to return is not display
ed.
        fmt.Println("RowCount: ", len(searchResponse.Rows))
    }
}
```

3.7.15. Nested query

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

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
Path	The path of the NESTED column. The path is similar to the tree structure. For example, news.title indicates the title subcolumn in the news column of the NESTED type.
Query	The query on the subcolumn in the NESTED column. The query can be of any query type.
ScoreMode	The value on which a score is calculated when multiple values exist for a column.

Examples

The NESTED column contains nested_1 and nested_2. You must search the col_nested.nested_1 column for data that matches "tablestore".

```
func NestedQuery(client *tablestore.TableStoreClient, 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 of the NESTED field.
        Query: &search.TermQuery{ // Construct the subquery of NestedQuery.
            FieldName: "col_nested.nested_1", // Set the field that is prefixed with col_ne
sted.
            Term:      "tablestore",          // Set the value you want to match.
        },
        ScoreMode: search.ScoreMode_Avg,
    }
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    // Return all columns.
    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))
    }
}
```

3.7.16. Sorting and pagination

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

Index presorting

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

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

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

Specify a sorting method

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

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

- ScoreSort

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

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

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

- PrimaryKeySort

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

  ```
  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 results based on the values of a specified column.

```
    // Sort the query results based on the values of Col_Long in descending order.
    searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.FieldSort{
                FieldName: "Col_Long",
                Order:     search.SortOrder_DESC.Enum(),
            },
        },
    })
```

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

```
searchQuery.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 results by geographical location.

```
    searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.GeoDistanceSort{
                FieldName: "location",   // Set the name of the geographical location fie
ld.
                Points:    []string{"40,-70"}, // Set the coordinate pairs of a central p
oint.
            },
        },
    })
```

Specify a pagination method

You can use Limit and Offset or Token for pagination when you obtain returned results.

- Use the Limit and Offset parameters

  When the total number of rows to obtain is smaller than 50,000, you can specify the Limit and Offset parameters for pagination. The sum of the Limit and Offset parameter values cannot exceed 50,000. The maximum value of Limit is 100.

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

If you use Limit and Offset for pagination but do not configure the Limit and Offset values, the default values are used. The default value of Limit is 10. The default value of Offset is 0.

```
searchQuery := search.NewSearchQuery()
searchQuery.SetLimit(10)
searchQuery.SetOffset(10)
```

- Use a token

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

  If Tablestore cannot complete reading data that meets the query condition, the server returns NextToken. You can use NextToken to continue reading the subsequent data.

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

  When you use the token, the sorting method is the same as that used in the previous request. Tablestore sorts data based on the IndexSort field by default or based on the method that you have specified. Therefore, you cannot set the sorting method if you use a token. You cannot set Offset when a token is used. Data is returned page by page, which results in a slow query.

  > **Notice** Search indexes of the NEST type do not support IndexSort. If you use a search index of the NEST type to query data and require pagination, you must specify the sorting method to return data in the query conditions. Otherwise, Tablestore does not return NextToken when all data that meets the query conditions is not completely read.

```
/**
 * 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 query condition is returned if the NextToken field value is
nil.
 */
func QueryRowsWithToken(client *tablestore.TableStoreClient, 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("IsAllSuccess: ", searchResponse.IsAllSuccess)
        fmt.Println("TotalCount: ", searchResponse.TotalCount)
        fmt.Println("RowsSize: ", len(rows))
        fmt.Println("RequestCount: ", requestCount)
    }
}
```

3.7.17. Exists query

Exists query is also called NULL query or NULL-value query. This query is used in sparse data to determine whether a column of a row exists. For example, you can query the rows in which the value of the address column is not empty.

> **Note**
>
> - If you want to check whether a column contains empty values, you must use ExistsQuery together with MustNotQueries of BoolQuery.
> - If one of the following conditions is met, the system considers that a column does not to exist. In this example, the city column is used.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the city column does not exist in the data table, the search index considers that the city column does not exist.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the value of the city column is an empty array in the data table ("city" = "[]"), the search index considers that the city column does not exist.

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the data table.
IndexName	The name of the search index.
Query	The query type. Set the value to ExistsQuery.
FieldName	The name of the column.

Examples

The following sample code shows how to query the rows in which the city column is not empty:

```
func ExitsQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.ExitsQuery{} // Set the query type to ExitsQuery.
    query.FieldName = "city" // Specify the name of the column that you want to query.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchRequest.SetSearchQuery(searchQuery)
    // Specify that all columns are returned.
    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 rows th
at meet the query conditions are returned.
    fmt.Println("TotalCount: ", searchResponse.TotalCount) // Specify that the total number
of rows that meet the query conditions instead of the number of returned rows is displayed.

    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))
    }
}
```

3.7.18. Collapse (distinct)

Prerequisites

- The TableStoreClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Usage notes

- If you use the collapse feature, you must implement pagination by specifying offset and limit instead of Token.
- If you aggregate and collapse a result set at the same time, the result set is aggregated before it is collapsed.
- If you collapse the query results, the total number of groups of the returned results is determined by the sum of the offset and size values. A maximum of 10,000 groups can be returned.

- The total number of rows returned in the results indicates the number of returned rows before you use the collapse feature. After the result set is collapsed, you cannot query the total number of the returned groups.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
Query	The query type.
Collapse	The collapse parameter, including FieldName. FieldName: the name of the column based on which the result set is collapsed. Only columns whose values are of the INTEGER, FLOATING-POINT and KEYWORD data types are supported.
Offset	The position from which the current query starts.
Limit	The maximum number of rows that the current query returns. To query only the number of matched rows without the data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without table data.

Examples

```
func QueryWithCollapse(client *tablestore.TableStoreClient, tableName string, indexName str
ing) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)       // Specify the name of the table.
    searchRequest.SetIndexName(indexName)       // Specify the name of the search index.
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(&search.MatchQuery{    // Specify the query conditions.
        FieldName:          "user_id",
        Text:               "00002",
    })
    searchQuery.SetCollapse(&search.Collapse{
        FieldName: "product_name",              // Collapse the result set based on the pro
duct_name value.
    })
    searchQuery.SetOffset(0)
    searchQuery.SetLimit(100)
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{ReturnAll:true}) // Specify to r
eturn all columns.
    searchRequest.SetSearchQuery(searchQuery)
    searchResponse, err := client.Search(searchRequest)   // Start the query.
    if err != nil {
        fmt.Println("Failed to search with error:", err)
        return
    }
    for _, row := range searchResponse.Rows {           // Display the rows returned.
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

3.7.19. Aggregation

You can call the Aggregation operation to obtain the minimum value, maximum value, sum, average, count and distinct count of rows, and rows in each group from an aggregation operation. You can also call the Aggregation operation to group results by field value, range, geographical location, or filter, and implement nested query. You can also use multiple aggregation features to perform complex queries.

Minimum value

The aggregation method that can be used to return the minimum value of a field. This method can be used in a similar manner as the SQL MIN function.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.

Parameter	Description
FieldName	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
Missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The price of each product is listed in the product table. Query the minimum price of t
he products produced in Zhejiang.
 * The equivalent SQL statement: SELECT min(column_price) FROM product where place_of_pro
duction = "Zhejiang".
 */
func min(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "Zhejiang"}).
            SetLimit(0).    // If you are concerned only with aggregate results and not s
pecific data, you can set limit to 0 to improve performance.
            Aggregation(search.NewMinAggregation("min_agg_1", "column_price").Missing(0.0
0)))
    searchResponse, err := client.Search(searchRequest)    // Execute the query.
    aggResults := searchResponse.AggregationResults        // Obtain all aggregate result
s.
    agg1, err := aggResults.Min("min_agg_1")        // Obtain the aggregate result of min
_agg_1.
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {        // Specify whether the aggregate result of min_agg_1 has
a value.
        fmt.Println(agg1.Value)    // Display the aggregate result.
    }
}
```

Maximum value

The aggregation method that can be used to return the maximum value of a field. This method can be used in a similar manner as the SQL MAX function.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
Missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The price of each product is listed in the product table. Query the maximum price of t
he products produced in Zhejiang.
 * The equivalent SQL statement: SELECT max(column_price) FROM product where place_of_pro
duction = "Zhejiang".
 */
func max(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "Zhejiang"}).
            SetLimit(0).    // If you are concerned only with aggregate results and not s
pecific data, you can set limit to 0 to improve performance.
            Aggregation(search.NewMaxAggregation("max_agg_1", "column_price").Missing(0.0
0)))
    searchResponse, err := client.Search(searchRequest)    // Execute the query.
    aggResults := searchResponse.AggregationResults        // Obtain all aggregate result
s.
    agg1, err := aggResults.Max("max_agg_1")        // Obtain the aggregate result of max
_agg_1.
    if err != nil {
        panic(err)
    }
    if agg1.if agg1.HasValue() {        // Specify whether the aggregate result of max_ag
g_1 has a value.
        fmt.Println(agg1.Value)    // Display the aggregate result.
    }
}
```

Sum

The aggregation method that can be used to return the sum of all values for a numeric field. This method can be used in a similar manner as the SQL SUM function.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
Missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The sales volume of each product is listed in the product table. Query the total numbe
r of the sold products that are produced in Zhejiang. The value of the missing parameter
is set to 10.
 * The equivalent SQL statement: SELECT sum(column_price) FROM product where place_of_pro
duction="Zhejiang".
 */
func sum(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "Zhejiang"}).
            SetLimit(0).    // If you are concerned only with aggregate results and not s
pecific data, you can set limit to 0 to improve performance.
            Aggregation(search.NewSumAggregation("sum_agg_1", "column_price").Missing(0.0
0)))
    searchResponse, err := client.Search(searchRequest)    // Execute the query.
    aggResults := searchResponse.AggregationResults        // Obtain all aggregate result
s.
    agg1, err := aggResults.Sum("sum_agg_1")        // Obtain the aggregate result of sum
_agg_1.
    if err != nil {
        panic(err)
    }
    fmt.Println(agg1.Value)    // Display the aggregate result.
}
```

Average value

The aggregation method that can be used to return the average of all values for a numeric field. This method is used in a similar manner as the SQL AVG function.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
Missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ○ If you do not specify a value for the missing parameter, the row is ignored. ○ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * The sales volume of each product is listed in the product table. Query the average pri
ce of the products produced in Zhejiang.
 * The equivalent SQL statement: SELECT avg(column_price) FROM product where place_of_pro
duction = "Zhejiang".
 */
func avg(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "Zhejiang"}).
            SetLimit(0).    // If you are concerned only with aggregate results and not s
pecific data, you can set limit to 0 to improve performance.
            Aggregation(search.NewAvgAggregation("avg_agg_1", "column_price").Missing(0.0
0)))
    searchResponse, err := client.Search(searchRequest)    // Execute the query.
    aggResults := searchResponse.AggregationResults        // Obtain all aggregate result
s.
    agg1, err := aggResults.Avg("avg_agg_1")        // Obtain the aggregate result of avg
_agg_1.
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {        // Specify whether the aggregate result of agg1 has a val
ue.
        fmt.Println(agg1.Value)    // Display the aggregate result.
    }
}
```

Count

The aggregation method that can be used to return the total number of values for a specified field or the total number of rows in a table. This method can be used in a similar manner as the SQL COUNT function.

Note You can use one of the following methods to query the total number of rows in a table or the total number of rows that match the query conditions:

- Use the count feature of aggregation. Set the count parameter to * in the request.
- Use the query feature to obtain the number of rows that match the query conditions. Set the setGetTotalCount parameter to true in the query. Use MatchAllQuery to obtain the total number of rows in a table.

You can use the name of a column as the value of the count expression to query the number of rows that contain the column in a table. This method is suitable for scenarios that involve sparse columns.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT data types are supported.

- Examples

```
/**
 * Punishment records of merchants are recorded in the merchant table. You can query the
number of merchants who are located in Zhejiang and have punishment records. If a merchan
t does not have punishment records, the specified field is empty for that merchant.
 * The equivalent SQL statement: SELECT count(column_history) FROM product where place_of
_production="Zhejiang".
 */
func count(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "Zhejiang"}).
            SetLimit(0).    // If you are concerned only with aggregate results and not s
pecific data, you can set limit to 0 to improve performance.
            Aggregation(search.NewCountAggregation("count_agg_1", "column_price")))
    searchResponse, err := client.Search(searchRequest)     // Execute the query.
    aggResults := searchResponse.AggregationResults        // Obtain all aggregate result
s.
    agg1, err := aggResults.Count("count_agg_1")        // Obtain the aggregate result of
count_agg_1.
    if err != nil {
        panic(err)
    }
    fmt.Println(agg1.Value)     // Display the aggregate result.
}
```

Distinct count

The aggregation method that can be used to return the number of distinct values for a field. This method can be used in a similar manner as the SQL COUNT(DISTINCT) function.

> **Note** The number of distinct values is an approximate number.
>
> - If the total number of rows before the distinct count feature is used is less than 10,000, the calculated result is an exact value.
> - If the total number of rows before the distinct count feature is used is greater than or equal to 100 million, the error rate is approximately 2%.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT data types are supported.
Missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ◦ If you do not specify a value for the missing parameter, the row is ignored. ◦ If you specify a value for the missing parameter, the value of this parameter is used as the field value of the row.

- Examples

```
/**
 * Query the number of distinct provinces from which the products come.
 * The equivalent SQL statement: SELECT count(distinct column_place) FROM product.
 */
func distinctCount(client *tablestore.TableStoreClient, tableName string, indexName strin
g) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "Zhejiang"}).
            SetLimit(0).    // If you are concerned only with aggregate results and not s
pecific data, you can set limit to 0 to improve performance.
            Aggregation(search.NewDistinctCountAggregation("distinct_count_agg_1", "colum
n_price").Missing(0.00)))
    searchResponse, err := client.Search(searchRequest)    // Execute the query.
    aggResults := searchResponse.AggregationResults        // Obtain all aggregate result
s.
    agg1, err := aggResults.DistinctCount("distinct_count_agg_1")        // Obtain the ag
gregate result of distinct_count_agg_1.
    if err != nil {
        panic(err)
    }
    fmt.Println(agg1.Value)    // Display the aggregate result.
}
```

Group by field value

The aggregation method that can be used to group query results based on field values. The values that are the same are grouped together. The identical value of each group and the number of identical values in each group are returned.

> **Note** The calculated number may be different from the actual number if the number of values in a group is very large.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the LONG, DOUBLE, BOOLEAN, and KEYWORD data types are supported.
Size	The number of returned groups.

Parameter	Description
GroupBySorters	The sorting rules for items in a group. By default, group items are sorted in descending order. If you configure multiple sorting rules, data is sorted based on the order in which the rules are added. Supported parameters: ○ Sort by value in ascending alphabetical order ○ Sort by value in descending alphabetical order ○ Sort by row count in ascending order ○ Sort by row count in descending order ○ Sort in ascending order of the values that are obtained from sub-aggregate results ○ Sort in descending order of the values that are obtained from sub-aggregate results
SubAggregation and SubGroupBy	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results. ○ Scenario Query the number of products in each category and the maximum and minimum product prices in each category. ○ Method Group query results by product category to obtain the number of products in each category. Then, add two sub-aggregations to obtain the maximum and minimum product prices in each category. ○ Examples ■ Fruits: The number is 5. The maximum price is 15. The minimum price is 3. ■ Toiletries: The number is 10. The maximum price is 98. The minimum price is 1. ■ Electronic devices: The number is 3. The maximum price is 8,699. The minimum price is 2,300. ■ Other products: The number is 15. The maximum price is 1,000. The minimum price is 80.

- Examples

```
/**
 * Query the number of products and the maximum and minimum product prices in each catego
ry.
 * Example of returned results: Fruits: 5. The minimum price is 3, and the maximum price
is 15. Toiletries: 10. The minimum price is 1, and the maximum price is 98. Electronic de
vices: 3. The minimum price is 2,300, and the maximum price is 8,699. Other products: 15.
The minimum price is 80,
 * and the maximum price is 1,000.
 */
func GroupByField(client *tablestore.TableStoreClient, tableName string, indexName string
) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    // Match all rows.
            SetLimit(0).
            GroupBy(search.NewGroupByField("group1", "column_type").    // Group by value
for a field in each category.
                SubAggregation(search.NewMinAggregation("min_price", "column_price")).
// The minimum product price in each category.
                SubAggregation(search.NewMaxAggregation("max_price", "column_price"))))
// The maximum product price in each category.
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    // Obtain all aggregate results.
    group, err := groupByResults.GroupByField("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    // Traverse the returned groups.
        // Display the value of each group and the number of rows in each group.
        fmt.Println("\tkey: ", item.Key, ", rowCount: ", item.RowCount)
        // Display the minimum prices.
        minPrice, _ := item.SubAggregations.Min("min_price")
        if minPrice.HasValue() {
            fmt.Println("\t\tmin_price: ", minPrice.Value)
        }
        // Display the maximum prices.
        maxPrice, _ := item.SubAggregations.Max("max_price")
        if maxPrice.HasValue() {
            fmt.Println("\t\tmax_price: ", maxPrice.Value)
        }
    }
}
```

Group by range

The aggregation method that can be used to group query results based on the value ranges of a field. Field values that are within a specified range are grouped together. The number of values in each range is returned.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
Range(fromInclusive float64, toExclusive float64)	The value ranges for grouping. The range can start from NegInf and end with Inf.
SubAggregation and SubGroupBy	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results. For example, after you group query results by sales volume and then by province, you can obtain the province that has the largest proportion of sales volume in a specified range. You must specify GroupByField in GroupByRange to run this query.

- Examples

```
/**
 * Query the sales of each range by grouping [NegInf,1000), [1000,5000), and [5000,Inf) w
hen you calculate the sales.
 */
func GroupByRange(client *tablestore.TableStoreClient, tableName string, indexName string
) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    // Match all rows.
            SetLimit(0).
            GroupBy(search.NewGroupByRange("group1", "column_number").
                Range(search.NegInf, 1000).
                Range(1000, 5000).
                Range(5000, search.Inf)))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    // Obtain all aggregate results.
    group, err := groupByResults.GroupByRange("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    // Traverse the returned groups.
        fmt.Println("\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)    /
/ Display the number of rows in each group.
    }
}
```

Group by geographical location

The aggregation method that can be used to group query results based on geographical locations to a central point. Query results in distances that are within a specified range are grouped together. The number of items in each range is returned.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
FieldName	The name of the field used to perform the aggregation operation. Only the GEOPOINT data type is supported.
CenterPoint(latitude float64, longitude float64)	The longitude and latitude of the central point. latitude indicates the latitude of the central point. longitude indicates the longitude of the central point.

Parameter	Description
Range(fromInclusive float64, toExclusive float64)	The ranges for grouping. Unit: meter. The range can start from NegInf and end with Inf.
SubAggregation and SubGroupBy	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results.

- Examples

```
/**
 * Group people based on their geographical locations to a Wanda Plaza to obtain the resu
lt of the number of people in each distance range. Group the distances based on ranges [N
egInf, 1000), [1000, 5000), [5000, Inf). The distances are measured in meter.
 */
func GroupByGeoDistance(client *tablestore.TableStoreClient, tableName string, indexName
string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    // Match all rows.
            SetLimit(0).
            GroupBy(search.NewGroupByGeoDistance("group1", "Col_GeoPoint", search.GeoPoin
t{Lat: 30.137817, Lon:120.08681}).
                Range(search.NegInf, 1000).
                Range(1000, 5000).
                Range(5000, search.Inf)))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    // Obtain all aggregate results.
    group, err := groupByResults.GroupByGeoDistance("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    // Traverse the returned groups.
        fmt.Println("\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)    /
/ Display the number of rows in each group.
    }
}
```

Group by filter

The aggregation method that can be used to filter the query results and group them together to obtain the number of results that match each filter. Results are returned in the order in which the filters are specified.

- Parameters

Parameter	Description
Name	The unique name of the aggregation operation. You can query the results of the aggregation operation based on the name.
Query	The filters for the query. Results are returned in the order in which the filters are added.
SubAggregation and SubGroupBy	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results.

- Examples

```
/**
 * Add the following three filters to obtain the number of items that match each filter:
The sales volume exceeds 100, the production place is Zhejiang, and the description conta
ins Hangzhou.
 */
func GroupByFilter(client *tablestore.TableStoreClient, tableName string, indexName strin
g) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    // Match all rows.
            SetLimit(0).
            GroupBy(search.NewGroupByFilter("group1").
                Query(&search.RangeQuery{
                    FieldName: "number",
                    From: 100,
                    IncludeLower: true}).
                Query(&search.TermQuery{
                    FieldName: "place",
                    Term:      "Zhejiang",
                }).
                Query(&search.MatchQuery{
                    FieldName: "description",
                    Text: "Hangzhou"
                })))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    // Obtain all aggregate results.
    group, err := groupByResults.GroupByFilter("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    // Traverse the returned groups.
        fmt.Println("\trowCount: ", item.RowCount)    // Display the number of rows in ea
ch group.
    }
}
```

Nesting

GroupBy supports nesting. You can perform sub-aggregation operations by using GroupBy.

You can use nesting to perform sub-aggregation operations in a group. For example, you can perform nesting aggregation operations up to two levels.

- GroupBy + SubGroupBy: Items are grouped by province and by city to obtain data for each city in each province.
- GroupBy + SubAggregation: Items are grouped by province to obtain the maximum value of a metric

for each province.

> **Note** To ensure high performance of queries and perform GroupBy operations, you can specify only a small number of levels for nesting. For more information, see Search index limits.

Examples

```
/**
 * Perform nesting-based aggregation.
 * Two Aggregations and one GroupByRange are added to the outermost level of GroupByField.
 */
func NestedSample(client *tablestore.TableStoreClient, tableName string, indexName string)
{
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    // Match all rows.
            SetLimit(0).
            GroupBy(search.NewGroupByField("group1", "field1").
                SubAggregation(search.NewMinAggregation("sub_agg1", "sub_field1")).
                SubAggregation(search.NewMaxAggregation("sub_agg2", "sub_field2")).
                SubGroupBy(search.NewGroupByRange("sub_group1", "sub_field3").
                    Range(search.NegInf, 3).
                    Range(3, 5).
                    Range(5, search.Inf))))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    // Obtain all aggregate results.
    group, err := groupByResults.GroupByField("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    // Traverse the returned groups.
        // Display the value of each group and the number of rows in each group.
        fmt.Println("\tkey: ", item.Key, ", rowCount: ", item.RowCount)
        // Obtain the aggregate results of sub_agg1.
        subAgg1, _ := item.SubAggregations.Min("sub_agg1")
        if subAgg1.HasValue() {
            fmt.Println("\t\tsub_agg1: ", subAgg1.Value)
        }
        // Obtain the aggregate results of sub_agg2.
        subAgg2, _ := item.SubAggregations.Max("sub_agg2")
        if subAgg2.HasValue() {
            fmt.Println("\t\tsub_agg2: ", subAgg2.Value)
        }
        // Obtain the aggregate results of sub_group1.
        subGroup, _ := item.SubGroupBys.GroupByRange("sub_group1")
        for _, item := range subGroup.Items {
            fmt.Println("\t\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)
        }
    }
}
```

Multiple aggregations

You can perform multiple aggregation operations.

> **Note** If you perform multiple complex aggregation operations at the same time, a long period of time may be required.

- Example 1

```
func MultipleAggregations(client *tablestore.TableStoreClient, tableName string, indexNam
e string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    // Match all rows.
            SetLimit(0).
            Aggregation(search.NewAvgAggregation("agg1", "Col_Long")).                // 
Calculate the average value for the Col_Long field.
            Aggregation(search.NewDistinctCountAggregation("agg2", "Col_Long")).    // Ca
lculate the number of distinct values for the Col_Long field.
            Aggregation(search.NewMaxAggregation("agg3", "Col_Long")).                // 
Calculate the maximum value for the Col_Long field.
            Aggregation(search.NewSumAggregation("agg4", "Col_Long")).                // 
Calculate the sum for the Col_Long field.
            Aggregation(search.NewCountAggregation("agg5", "Col_Long")))            // Ca
lculate the number of rows that contain the Col_Long field.
    // Return all columns.
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll: true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    aggResults := searchResponse.AggregationResults    // Obtain all aggregate results.
    // Obtain the average value obtained from the aggregation operation.
    agg1, err := aggResults.Avg("agg1")        // Obtain the average value for agg1.
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {                            // Specify whether the aggregate resu
lt of agg1 has a value.
        fmt.Println("(avg) agg1: ", agg1.Value)    // Display the average value for the C
ol_Long field.
    } else {
        fmt.Println("(avg) agg1: no value")        // Display rows that contain no values
for the Col_Long field.
    }
    // Obtain the distinct count obtained from the aggregation operations.
    agg2, err := aggResults.DistinctCount("agg2")    // Obtain the number of distinct val
ues for agg2.
```

```
    if err != nil {
        panic(err)
    }
    fmt.Println("(distinct) agg2: ", agg2.Value)        // Display the number of distinct
values for the Col_Long field.
    // Display the maximum value obtained from the aggregation operation.
    agg3, err := aggResults.Max("agg3")        // Obtain the maximum value for agg3.
    if err != nil {
        panic(err)
    }
    if agg3.HasValue() {
        fmt.Println("(max) agg3: ", agg3.Value)    // Display the maximum value for the C
ol_Long field.
    } else {
        fmt.Println("(max) agg3: no value")        // Display rows that contain no values
for the Col_Long field.
    }
    // Obtain the sum calculated from the results of the aggregation operation.
    agg4, err := aggResults.Sum("agg4")        // Obtain the sum for agg4.
    if err != nil {
        panic(err)
    }
    fmt.Println("(sum) agg4: ", agg4.Value)        // Display the sum of the Col_Long fie
ld values.
    // Obtain the aggregate result of the count.
    agg5, err := aggResults.Count("agg5")        // Obtain the count for agg5.
    if err != nil {
        panic(err)
    }
    fmt.Println("(count) agg6: ", agg5.Value)    // Display the count of rows that contai
n values for the Col_Long field.
}
```

- Example 2

```
func MultipleAggregationsAndGroupBysSample(client *tablestore.TableStoreClient, tableName
string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    // Specify the name of the data table.
        SetIndexName(indexName).    // Specify the name of the search index.
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    // Match all rows.
            SetLimit(0).
            Aggregation(search.NewAvgAggregation("agg1", "Col_Long")).                //
Calculate the average value for the Col_Long field.
            Aggregation(search.NewDistinctCountAggregation("agg2", "Col_Long")).    // Ca
lculate the number of distinct values for the Col_Long field.
            Aggregation(search.NewMaxAggregation("agg3", "Col_Long")).                //
Calculate the maximum value for the Col_Long field.
            GroupBy(search.NewGroupByField("group1", "Col_Keyword").    // Perform the Gr
oupByField operation for the Col_Keyword field.
                GroupBySorters([]search.GroupBySorter{}).    // Specify the order in whic
h the groups are returned.
                Size(2).                                    // Return only the first two grou
```

```
                Size(2).                                    // Return only the first two grou
ps.
                SubAggregation(search.NewAvgAggregation("sub_agg1", "Col_Long")).    // P
erform the sub-aggregation operation for each group.
                SubGroupBy(search.NewGroupByField("sub_group1", "Col_Keyword2"))).    //
Perform the sub-aggregation operation for each group.
            GroupBy(search.NewGroupByRange("group2", "Col_Long").        // Perform the G
roupByRange operation for the Col_Long field.
                Range(search.NegInf, 3).            // The first group contains rows whos
e values in the Col_Long field are within the (NegInf, 3) range.
                Range(3, 5).            // The second group contains rows whose values in
the Col_Long field are within the [3, 5) range.
                Range(5, search.Inf)))            // The third group contains rows whose
values in the Col_Long field are within the [5, Inf) range.
    // Return all columns.
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    aggResults := searchResponse.AggregationResults    // Obtain all aggregate results.
    // Obtain the average value obtained from the aggregation operation.
    agg1, err := aggResults.Avg("agg1")        // Obtain the average value for agg1.
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {                           // Specify whether the aggregate resu
lt of agg1 has a value.
        fmt.Println("(avg) agg1: ", agg1.Value)    // Display the average value for the C
ol_Long field.
    } else {
        fmt.Println("(avg) agg1: no value")        // Display rows that contain no values
for the Col_Long field.
    }
    // Obtain the distinct count obtained from the aggregation operations.
    agg2, err := aggResults.DistinctCount("agg2")    // Obtain the number of distinct val
ues for agg2.
    if err != nil {
        panic(err)
    }
    fmt.Println("(distinct) agg2: ", agg2.Value)        // Display the number of distinct
values for the Col_Long field.
    // Display the maximum value obtained from the aggregation operation.
    agg3, err := aggResults.Max("agg3")        // Obtain the maximum value for agg3.
    if err != nil {
        panic(err)
    }
    if agg3.HasValue() {
        fmt.Println("(max) agg3: ", agg3.Value)    // Display the maximum value for the C
ol_Long field.
    } else {
        fmt.Println("(max) agg3: no value")        // Display rows that contain no values
for the Col_Long field.
    }
    groupByResults := searchResponse.GroupByResults    // Obtain all aggregate results.
```

```
    // Obtain the GroupByField values from the results of the aggregation operation.
    group1, err := groupByResults.GroupByField("group1")    // Obtain the GroupBy result
of group1, which is of the GroupByField type.
    if err != nil {
        panic(err)
    }
    fmt.Println("group1: ")
    for _, item := range group1.Items {    // Traverse the returned groups.
        //item
        fmt.Println("\tkey: ", item.Key, ", rowCount: ", item.RowCount)    // Display the
number of rows in each group.
        // Obtain the sub-aggregation result of the average value.
        subAgg1, err := item.SubAggregations.Avg("sub_agg1")    // Obtain the sub-aggrega
tion result of sub_agg1.
        if err != nil {
            panic(err)
        }
        if subAgg1.HasValue() {    // If sub_agg1 obtains the average value for the Col_L
ong field, the HasValue() parameter value is true.
            fmt.Println("\t\tsub_agg1: ", subAgg1.Value)    // Display the average value
for the Col_Long field.
        }
        // Obtain the GroupByField values from the aggregation operation.
        subGroup1, err := item.SubGroupBys.GroupByField("sub_group1")    // Obtain the su
b-GroupBy result of sub_group1.
        if err != nil {
            panic(err)
        }
        fmt.Println("\t\tsub_group1")
        for _, subItem := range subGroup1.Items {    // Traverse the sub-aggregate result
s for sub_group1.
            fmt.Println("\t\t\tkey: ", subItem.Key, ", rowCount: ", subItem.RowCount)
// Display the number of rows for sub_group1.
            tablestore.Assert(subItem.SubAggregations.Empty(), "")
            tablestore.Assert(subItem.SubGroupBys.Empty(), "")
        }
    }
    // Obtain the GroupByRange values from the results of the aggregation operation.
    group2, err := groupByResults.GroupByRange("group2")    // Obtain the GroupBy result
of group2, which is of the GroupByRange type.
    if err != nil {
        panic(err)
    }
    fmt.Println("group2: ")
    for _, item := range group2.Items {    // Traverse the returned groups.
        fmt.Println("\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)    /
/ Display the number of rows in each group.
    }
}
```

3.7.20. Parallel scan

In scenarios where you are not concerned with the order of the query results, you can use parallel scan to obtain query results in parallel.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A table is created. Data is written to the table.
- A search index is created on the table. For more information, see Create search indexes.

Parameters

Parameter		Description
TableName		The name of the table.
IndexName		The name of the search index.
ScanQuery	Query	The query statement for the search index. The operation supports term query, fuzzy query, range query, geo query, and nested query, which is similar to those of the Search operation.
	Limit	The maximum number of rows that can be returned at a time.
	MaxParallel	The maximum number of concurrent requests. The maximum number of concurrent requests varies based on the data volume. The larger volume of data requires more concurrent requests. You can use the ComputeSplits operation to query the maximum number of concurrent requests.
	CurrentParallelID	The ID of the concurrent request. Valid values: [0, MaxParallel).
	Token	The token used for pagination. The result of the parallel scan request contains the token for the next page. You can use the token to query data for the following page.

Parameter		Description
	AliveTime	The validity period of the current parallel scan task. The value is also the validity period of the token. We recommend that you use the default value. Default value: 60. Unit: seconds. If the next request is not initiated within the validity period, no more data can be queried. The validity period of the token is refreshed each time you send a request. **Note** The server uses the asynchronous method to process expired tasks. The current task does not expire within the validity period. However, Tablestore does not guarantee that the task expires after the validity period.
ColumnsToGet		Parallel scan can be used only to query indexed columns in search indexes. You must set store to true when you create a search index.
SessionId		The session ID of the parallel scan task. You can call the ComputeSplits operation to create a session and obtain the maximum number of concurrent requests supported in the current task.

Examples

The following code provides examples on how to scan data by sending a single request or by using multiple threads at a time:

- Scan data by sending a single request

```
/**
 * Send a single parallel scan request to scan
data.
 */
func ParallelScanSingleConcurrency(client *tablestore.TableStoreClient, tableName string,
indexName string) {
        computeSplitsResp, err := computeSplits(client, tableName, indexName)
        if err != nil {
                fmt.Printf("%#v", err)
                return
        }

        query := search.NewScanQuery().SetQuery(&search.MatchAllQuery{}).SetLimit(2)

        req := &tablestore.ParallelScanRequest{}
        req.SetTableName(tableName).
                SetIndexName(indexName).
                SetColumnsToGet(&tablestore.ColumnsToGet{ReturnAllFromIndex: false}).
                SetScanQuery(query).
                SetSessionId(computeSplitsResp.SessionId)

        res, err := client.ParallelScan(req)
        if err != nil {
                fmt.Printf("%#v", err)
                return
        }

        total := len(res.Rows)
        for res.NextToken != nil {
                req.SetScanQuery(query.SetToken(res.NextToken))
                res, err = client.ParallelScan(req)
                if err != nil {
                        fmt.Printf("%#v", err)
                        return
                }

                total += len(res.Rows) //process rows each loop
        }
        fmt.Println("total: ", total)
}
```

- Use concurrent threads to scan data

```
/**
 * Send multiple parallel scan requests to scan
data.
 */
func ParallelScanMultiConcurrency(client *tablestore.TableStoreClient, tableName string,
indexName string) {
        computeSplitsResp, err := computeSplits(client, tableName, indexName)
        if err != nil {
                fmt.Printf("%#v", err)
                return
        }
```

```
        var wg sync.WaitGroup
        wg.Add(int(computeSplitsResp.SplitsSize))

        for i := int32(0); i < computeSplitsResp.SplitsSize; i++ {
                current := i
                go func() {
                        defer wg.Done()
                        query := search.NewScanQuery().
                                SetQuery(&search.MatchAllQuery{}).
                                SetCurrentParallelID(current).
                                SetMaxParallel(computeSplitsResp.SplitsSize).
                                SetLimit(2)

                        req := &tablestore.ParallelScanRequest{}
                        req.SetTableName(tableName).
                                SetIndexName(indexName).
                                SetColumnsToGet(&tablestore.ColumnsToGet{ReturnAllFromInd
ex: false}).
                                SetScanQuery(query).
                                SetSessionId(computeSplitsResp.SessionId)

                        res, err := client.ParallelScan(req)
                        if err != nil {
                                fmt.Printf("%#v", err)
                                return
                        }

                        total := len(res.Rows)
                        for res.NextToken != nil {
                                req.SetScanQuery(query.SetToken(res.NextToken))
                                res, err = client.ParallelScan(req)
                                if err != nil {
                                        fmt.Printf("%#v", err)
                                        return
                                }

                                total += len(res.Rows) //process rows each loop
                        }
                        fmt.Println("total: ", total)
                }()
        }
        wg.Wait()
}
```

3.8. Secondary Index

3.8.1. Global secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created. The value of TimeToLive is -1. The value of MaxVersions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table for an existing data table.

> **Note** You can call the CreateTable operation to create one or more index tables when you create a data table. For more information, see Create a data table.

- Parameters

Parameter	Description
MainTableName	The name of the data table.
IndexMeta	The schema information of the index table. The schema information contains the following items: ○ IndexName: the name of the index table. ○ PrimaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ○ DefinedColumns: the attribute columns of the index table. The attribute columns are a combination of predefined columns of the data table. ○ IndexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If IndexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. When you use a global secondary index, Tablestore automatically synchronizes the columns to be indexed and the data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If IndexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table.

Parameter	Description
IncludeBaseData	Specifies whether to include the existing data from the data table in the index table. When IncludeBaseData is set to true, the index table contains the existing data in the data table. When IncludeBaseData is set to false, the index table does not contain the existing data in the data table.

- Examples

  The following code provides an example on how to create an index table on a data table. The data table contains primary key columns pk1 and pk2. The index table contains primary key column definedcol1 and attribute column definedcol2.

```
func CreateGlobalIndexSample(client *tablestore.TableStoreClient, tableName string) {
    indexMeta := new(tablestore.IndexMeta) // Create IndexMeta for the index table.
    indexMeta.AddPrimaryKeyColumn("definedcol1") // Specify the definedcol1 column of the
data table as the primary key column of the index table.
    indexMeta.AddDefinedColumn("definedcol2") // Specify the definedcol2 column of the da
ta table as the attribute column of the index table.
    indexMeta.IndexName = "indexSample"
    indexReq := &tablestore.CreateIndexRequest{
        MainTableName:tableName, // Specify the data table to create the index table on t
he data table.
        IndexMeta: indexMeta,
        IncludeBaseData: true, // Specify that the index table contains the existing data
in the data table.
    }
    /**
      You can set the IncludeBaseData parameter to true to synchronize existing data in a
data table after an index table is created. Then, you can query all data from the index t
able.
      The amount of time required to synchronize data to the index table is determined by
the amount of data in the data table.
    */
    resp, err := client.CreateIndex(indexReq)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create index finished", resp)
    }
}
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

When you call the GetRow operation to read data from an index table, take note of the following items:

- You must set TableName to the name of the index table.
- Tablestore automatically adds the primary key columns that are not configured in the indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from an index table

  For more information, see Multi-row operations.

  - Parameters

    When you call the GetRange operation to read data from an index table, take note of the following items:

    - You must set TableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

  - Examples

```
func GetRangeFromIndex(client *tablestore.TableStoreClient, indexName string) {
    getRangeRequest := &tablestore.GetRangeRequest{}
    rangeRowQueryCriteria := &tablestore.RangeRowQueryCriteria{}
    rangeRowQueryCriteria.TableName = indexName
    startPK := new(tablestore.PrimaryKey)
    startPK.AddPrimaryKeyColumnWithMinValue("definedcol1") / Specify the first primary
key column of the index table.
    startPK.AddPrimaryKeyColumnWithMinValue("pk1") // Specify the third primary key col
umn of the index table, which is the primary key column that is automatically added fro
m the data table.
    startPK.AddPrimaryKeyColumnWithMinValue("pk2") // Specify the third primary key col
umn of the index table, which is the primary key column that is automatically added fro
m the data table.
    endPK := new(tablestore.PrimaryKey)
    endPK.AddPrimaryKeyColumnWithMaxValue("definedcol1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk2")
    rangeRowQueryCriteria.StartPrimaryKey = startPK
    rangeRowQueryCriteria.EndPrimaryKey = endPK
    rangeRowQueryCriteria.Direction = tablestore.FORWARD
    rangeRowQueryCriteria.MaxVersion = 1
    rangeRowQueryCriteria.Limit = 10
    getRangeRequest.RangeRowQueryCriteria = rangeRowQueryCriteria
    getRangeResp, err := client.GetRange(getRangeRequest)
    fmt.Println("get range result is ", getRangeResp)
    for {
        if err != nil {
            fmt.Println("get range failed with error:", err)
        }
        if len(getRangeResp.Rows) > 0 {
            for _, row := range getRangeResp.Rows {
                fmt.Println("range get row with key", row.PrimaryKey.PrimaryKeys[0].Val
ue, row.PrimaryKey.PrimaryKeys[1].Value, row.PrimaryKey.PrimaryKeys[2].Value)
            }
            if getRangeResp.NextStartPrimaryKey == nil {
                break
            } else {
                fmt.Println("next pk is :", getRangeResp.NextStartPrimaryKey.PrimaryKey
s[0].Value, getRangeResp.NextStartPrimaryKey.PrimaryKeys[1].Value, getRangeResp.NextSta
rtPrimaryKey.PrimaryKeys[2].Value)
                getRangeRequest.RangeRowQueryCriteria.StartPrimaryKey = getRangeResp.Ne
xtStartPrimaryKey
                getRangeResp, err = client.GetRange(getRangeRequest)
            }
        } else {
            break
        }
        fmt.Println("continue to query rows")
    }
    fmt.Println("putrow finished")
}
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
MainTableName	The name of the data table.
IndexName	The name of the index table.

- Examples

```
func DeleteIndex(client *tablestore.TableStoreClient, tableName string, indexName string)
{
    deleteIndex := &tablestore.DeleteIndexRequest{ MainTableName:tableName, IndexName: in
dexName }
    resp, err := client.DeleteIndex(deleteIndex)
    if err != nil {
        fmt.Println("Failed to delete index:", err)
    } else {
        fmt.Println("drop index finished", resp)
    }
```

3.8.2. Local secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created. The value of TimeToLive is -1. The value of MaxVersions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table for an existing data table.

> **Note** You can call the CreateTable operation to create one or more index tables when you create a data table. For more information, see Create a data table.

- Parameters

Parameter	Description
MainTableName	The name of the data table.

Parameter	Description
IndexMeta	The schema information of the index table. The schema information contains the following items: ◦ IndexName: the name of the index table. ◦ PrimaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ◦ DefinedColumns: the attribute columns of the index table. The attribute columns are a combination of predefined columns of the data table. ◦ IndexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If IndexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. When you use a global secondary index, Tablestore automatically synchronizes the columns to be indexed and the data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If IndexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table.
IncludeBaseData	Specifies whether to include the existing data from the data table in the index table. When IncludeBaseData is set to true, the index table contains the existing data in the data table. When IncludeBaseData is set to false, the index table does not contain the existing data in the data table.

- Examples

  The following code provides an example on how to create an index table for a data table. The data table contains primary key columns pk1 and pk2. The index table contains primary key columns pk1 and definedcol1 and attribute column definedcol2.

```
func CreateGlobalIndexSample(client *tablestore.TableStoreClient, tableName string) {
    indexMeta := new(tablestore.IndexMeta) // Create IndexMeta for the index table.
    indexMeta.AddPrimaryKeyColumn("pk1") /// The first primary key column of an index tab
le must be the same as that of the corresponding data table.
    indexMeta.AddPrimaryKeyColumn("definedcol1") // Specify the definedcol1 column of the
data table as the primary key column of the index table.
    indexMeta.AddDefinedColumn("definedcol2") // Specify the definedcol2 column of the da
ta table as the attribute column of the index table.
    indexMeta.IndexType = IT_LOCAL_INDEX // Set the index type to IT_LOCAL_INDEX.
    indexMeta.IndexName = "indexSample"
    indexReq := &tablestore.CreateIndexRequest{
        MainTableName:tableName, // Specify the data table to create the index table on t
he data table.
        IndexMeta: indexMeta,
        IncludeBaseData: true, // Specify that the index table contains the existing data
in the data table.
    }
    /**
      You can set the IncludeBaseData parameter to true to synchronize existing data in a
data table after an index table is created. Then, you can query all data from the index t
able.
      The amount of time required to synchronize data to the index table is determined by
the amount of data in the data table.
    */
    resp, err := client.CreateIndex(indexReq)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create index finished", resp)
    }
}
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  When you call the GetRow operation to read data from an index table, take note of the following items:

  - You must set TableName to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not configured in the indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from an index table

  For more information, see Multi-row operations.

- Parameters

  When you call the GetRange operation to read data from an index table, take note of the following items:

  - You must set TableName to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Examples

```
func GetRangeFromIndex(client *tablestore.TableStoreClient, indexName string) {
    getRangeRequest := &tablestore.GetRangeRequest{}
    rangeRowQueryCriteria := &tablestore.RangeRowQueryCriteria{}
    rangeRowQueryCriteria.TableName = indexName
    startPK := new(tablestore.PrimaryKey)
    startPK.AddPrimaryKeyColumnWithMinValue("pk1") // Specify the first primary key col
umn of the index table.
    startPK.AddPrimaryKeyColumnWithMinValue("definedcol1") // Specify the second primar
y key column of the index table.
    startPK.AddPrimaryKeyColumnWithMinValue("pk2") // Specify the third primary key col
umn of the index table, which is the primary key column that is automatically added fro
m the data table.
    endPK := new(tablestore.PrimaryKey)
    endPK.AddPrimaryKeyColumnWithMaxValue("pk1")
    endPK.AddPrimaryKeyColumnWithMaxValue("definedcol1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk2")
    rangeRowQueryCriteria.StartPrimaryKey = startPK
    rangeRowQueryCriteria.EndPrimaryKey = endPK
    rangeRowQueryCriteria.Direction = tablestore.FORWARD
    rangeRowQueryCriteria.MaxVersion = 1
    rangeRowQueryCriteria.Limit = 10
    getRangeRequest.RangeRowQueryCriteria = rangeRowQueryCriteria
    getRangeResp, err := client.GetRange(getRangeRequest)
    fmt.Println("get range result is ", getRangeResp)
    for {
        if err != nil {
            fmt.Println("get range failed with error:", err)
        }
        if len(getRangeResp.Rows) > 0 {
            for _, row := range getRangeResp.Rows {
                fmt.Println("range get row with key", row.PrimaryKey.PrimaryKeys[0].Val
ue, row.PrimaryKey.PrimaryKeys[1].Value, row.PrimaryKey.PrimaryKeys[2].Value)
            }
            if getRangeResp.NextStartPrimaryKey == nil {
                break
            } else {
                fmt.Println("next pk is :", getRangeResp.NextStartPrimaryKey.PrimaryKey
s[0].Value, getRangeResp.NextStartPrimaryKey.PrimaryKeys[1].Value, getRangeResp.NextSta
rtPrimaryKey.PrimaryKeys[2].Value)
                getRangeRequest.RangeRowQueryCriteria.StartPrimaryKey = getRangeResp.Ne
xtStartPrimaryKey
                getRangeResp, err = client.GetRange(getRangeRequest)
            }
        } else {
            break
        }
        fmt.Println("continue to query rows")
    }
    fmt.Println("putrow finished")
}
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
MainTableName	The name of the data table.
IndexName	The name of the index table.

- Examples

```
func DeleteIndex(client *tablestore.TableStoreClient, tableName string, indexName string)
{
    deleteIndex := &tablestore.DeleteIndexRequest{ MainTableName:tableName, IndexName: in
dexName }
    resp, err := client.DeleteIndex(deleteIndex)
    if err != nil {
        fmt.Println("Failed to delete index:", err)
    } else {
        fmt.Println("drop index finished", resp)
    }
```

3.9. Tunnel Service

3.9.1. Install Tunnel Service

This topic describes how to install Tunnel Service.

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 the dependencies.
  - Install the dep tool.
  - dep ensure -v
- You can also 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/smartystreets/goconvey/convey
go get github.com/golang/mock/gomock
go get gopkg.in/natefinch/lumberjack.v2
```

3.9.2. Quick start

Tablestore SDK for Go allows you to use Tunnel Service. Before you use Tunnel Service, you must understand the usage notes.

Usage notes

- By default, TunnelWorkerConfig starts the thread pool to read and process data. A single server starts multiple TunnelWorkers. We recommend that TunnelWorkers share the same TunnelWorkerConfig.
- When you create a differential tunnel to consume 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 consume 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 you estimate that the tunnel cannot consume full data within seven days, submit a ticket or join DingTalk group 23307953 to request technical support.
- TunnelWorker requires warm-up time during initialization. The HeartbeatInterval parameter in TunnelWorkerConfig determines the warm-up time. You can use the time method in TunnelWorkerConfig to set this parameter. The default value is 30s. The minimum value is 5s.
- When the mode switches from the full channel to the incremental channel, the full channel is closed and the incremental channel is started. This process requires the time for initialization. The HeartbeatInterval parameter determines the initialization time.
- When the TunnelWorker client is abnormally shut down due to an exceptional exit or manual termination, TunnelWorker automatically recycles resources in the following ways: 1. Release the thread pool. 2. Automatically call the shutdown method that you have registered for the Channel class. 3. Shut down the tunnel.

Experience Tunnel Service

1. Initialize a Tunnel client instance.

```
// Set endpoint to the endpoint of the Tablestore instance. Example: https://instance.c
n-hangzhou.ots.aliyun.com.
// Set the name of the instance.
// Set accessKeyId and accessKeySecret to the AccessKey ID and AccessKey secret used to
access Tablestore.
tunnelClient := tunnel.NewTunnelClient(endpoint, instance,
   accessKeyId, accessKeySecret)
```

2. Create a tunnel.

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

3. Customize the data consumption callback function to start automatic data consumption.

```
// Customize the data consumption 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 t
o the SimpleProcessFactory operation. Configure the TunnelWorkerConfig operation.
workConfig := &tunnel.TunnelWorkerConfig{
   ProcessorFactory: &tunnel.SimpleProcessFactory{
      CustomValue: "user custom interface{} value",
      ProcessFunc: exampleConsumeFunction,
   },
}
// Use TunnelDaemon to continuously consume the specified tunnel.
daemon := tunnel.NewTunnelDaemon(tunnelClient, tunnelId, workConfig)
log.Fatal(daemon.Run())
```

3.9.3. Configuration items

This topic describes the configuration items of Tunnel Service.

tunnel client

When you initialize a tunnel client, you can customize client configurations by using the NewTunnelClientWithConfig operation. 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 timeout period for the server to receive an HTTP request.
      RequestTimeout:      30 * time.Second,
      // Specify http.DefaultTransport.
      Transport:           http.DefaultTransport,
}
```

Worker

The TunnelWorkerConfig operation contains the configurations required for a worker. The ProcessorFactory field is required. If other parameters are unspecified, their default values are used.

```
type TunnelWorkerConfig struct {
   // The heartbeat timeout periods for the worker and Tunnel Service are the same. The def
ault value is retained.
   HeartbeatTimeout  time.Duration
   // Set the frequency at which worker sends heartbeat information. The default value is r
etained.
   HeartbeatInterval time.Duration
   // Set the operation that is used to establish connections for consumption of data expor
ted by using Tunnel Service. The default value is retained.
   ChannelDialer     ChannelDialer
   // Generate the specific processor for the connection. The callback function is typicall
y used to initialize the SimpleProcessFactory operation.
   ProcessorFactory ChannelProcessorFactory
   // Configure ZAP logs. The default value is DefaultLogConfig.
   LogConfig      *zap.Config
   // Configure log rotation for ZAP. The default value is DefaultSyncer.
   LogWriteSyncer zapcore.WriteSyncer
}
```

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

```
type SimpleProcessFactory struct {
   // Define and configure parameters. User-defined information is passed to the ChannelCon
text parameter contained in ProcessFunc and ShutdownFunc.
   CustomValue interface{}
   // Set the interval to receive information of two consecutive checkpoints. The worker re
cords the interval. When CpInterval is set to a value smaller than or equal to 0, DefaultCh
eckpointInterval 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 dat
a 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, DefaultLogConfig is used to initialize l
ogger.
   Logger *zap.Logger
}
```

Log management

The following examples describe the default configurations of logs and log rotation:

- Default configurations of logs

```
// By default, DefaultLogConfig is used in the TunnelWorkerConfig and SimpleProcessFactor
y operations to configure logs.
var DefaultLogConfig = zap.Config{
   Level:       zap.NewAtomicLevelAt(zap.InfoLevel),
   Development: false,
   Sampling: &zap.SamplingConfig{
      Initial:    100,
      Thereafter: 100,
   },
   Encoding: "json",
   EncoderConfig: zapcore.EncoderConfig{
      TimeKey:        "ts",
      LevelKey:       "level",
      NameKey:        "logger",
      CallerKey:      "caller",
      MessageKey:     "msg",
      StacktraceKey:  "stacktrace",
      LineEnding:     zapcore.DefaultLineEnding,
      EncodeLevel:    zapcore.LowercaseLevelEncoder,
      EncodeTime:     zapcore.ISO8601TimeEncoder,
      EncodeDuration: zapcore.SecondsDurationEncoder,
      EncodeCaller:   zapcore.ShortCallerEncoder,
   },
}
```

- Default configurations of log rotation

```
// By default, DefaultSyncer is used in the TunnelWorkerConfig and SimpleProcessFactory o
perations to configure log rotation.
var DefaultSyncer = zapcore.AddSync(&lumberjack.Logger{
   // Set the path of logs.
   Filename:   "tunnelClient.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,
})
```

3.9.4. Create tunnels

You can call the CreateTunnel operation to create one or more tunnels for a table. When you create a tunnel, you must specify the table name, tunnel name, and tunnel type.

Request parameters

Parameter	Description
TableName	The name of the table for which you wan to create a tunnel.

Parameter	Description
TunnelName	The tunnel name.
Type	The type of the tunnel. Valid values: BaseData, Stream, and BaseAndStream.

Response parameters

Parameter	Description
TunnelId	The channel ID.
ResponseInfo	Some other fields returned, including the RequestId field in the request. RequestId uniquely identifies the request.

Examples

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

3.9.5. Query the information of a tunnel

You can call the DescribeTunnel operation to obtain the information of channels in a specified tunnel. A channel corresponds to a shard for the Stream operation of Tablestore.

Request parameters

Parameter	Description
TableName	The name of the table for which you want to query the tunnel information.
TunnelName	The tunnel name.

Response parameters

Parameter	Description
TunnelRPO	The last time when the tunnel consumes incremental data. The time equals the point in time when the slowest channel in the tunnel consumes data. Default value: January 1, 1970 (UTC).
List<TunnelInfo>	The list of the tunnel, including the following information: • TunnelId: the ID of the tunnel. • TunnelName: the name of the tunnel. • TunnelType: the type of the tunnel. Valid values: BaseData, Stream, and BaseAndStream. • TableName: the name of the table for which the tunnel is configured. • InstanceName: the name of the instance for which the tunnel is configured. • Stage: the stage at which the tunnel is located. Valid values: InitBaseDataAndStreamShard, ProcessBaseData, and ProcessStream. • Expired: indicates whether the data is expired. If true is returned, contact Tablestore technical support by using DingTalk in a timely manner.
List<ChannelInfo>	The list of channels in the tunnel, including the following information: • ChannelId: the ID of the channel. • ChannelType: the type of the channel. Valid values: Base and Stream. • ChannelStatus: the status of the channel. Valid values: WAIT, OPEN, CLOSING, CLOSE, and TERMINATED. • ClientId: the ID of the tunnel client. By default, the client ID concatenates the hostname of the client and a random string. The client hostname can be customized by TunnelWorkerConfig. • ChannelRPO: the last time when a channel consumes incremental data. Default value: January 1, 1970 (UTC). This parameter is unavailable for the full data consumption.
ResponseInfo	Some other fields returned, including the RequestId field in the request. RequestId uniquely identifies the request.

Examples

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

3.9.6. Delete tunnels

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

Request parameters

Parameter	Description
TableName	The name of the table for which you want to delete the tunnel.
TunnelName	The tunnel name.

Response parameters

Parameter	Description
ResponseInfo	Some other fields returned, including the RequestId in the request. RequestId uniquely identifies the request.

Examples

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

3.10. SQL query

3.10.1. Overview

This topic describes the operations that are supported for Tablestore SDK for Go by executing SQL statements.

> **Notice** The latest version of Tablestore SDK for Go supports the SQL query feature. Before you use the SQL query feature, make sure that the latest version of Tablestore SDK for Go is obtained.

Operation	Description
Create mapping tables	You can execute the create table statement to create a mapping table for an existing data table or search index.
Delete mapping tables	You can execute the drop mapping table statement to delete mapping tables that are created for tables.
List the names of tables	You can execute the show tables statement to list the names of tables in the current database.
Query information about tables	You can execute the describe statement to query information about tables, such as the field names and field types.
Query index information about tables	You can execute the show index statement to query the index information about tables.
Query data	You can execute the select statement to query data in a table.

3.10.2. Create mapping tables

You can execute the create table statement to create a mapping table for an existing data table or search index.

> **Note** For more information about the create table statement, see Create mapping tables for tables.

Prerequisites

A Tablestore client is initialized. For more information, see Initialization.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `create table test_table (pk varchar(1024), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primary key(pk))` statement to create a mapping table for the table named test_table.

```
func createMappingTable(client *tablestore.TableStoreClient) {
    // Create a SQL request.
    request := &tablestore.SQLQueryRequest{Query: "create table test_table (pk varchar(1024
), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primar
y key(pk))"}
    // Obtain the response to the SQL request.
    response, err := client.SQLQuery(request)
}
```

3.10.3. Delete mapping tables

You can execute the drop mapping table statement to delete mapping tables that are created for tables.

> **Note** For more information about the drop mapping table statement, see Delete mapping tables.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `drop mapping table test_table` statement to delete the mapping tables that are created for the table named test_table.

```
func dropMappingTable(client *tablestore.TableStoreClient) {
    // Create a SQL request.
    request := &tablestore.SQLQueryRequest{Query: "drop mapping table test_table"}
    // Obtain the response to the SQL request.
    response, err := client.SQLQuery(request)
}
```

3.10.4. List the names of tables

You can execute the show tables statement to list the names of tables in the current database.

> **Note** For more information about the show tables statement, see List table names.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `show tables` statement to list the names of tables.

```
func showTable(client *tablestore.TableStoreClient) {
    // Create a SQL request.
    request := &tablestore.SQLQueryRequest{Query: "show tables"}
    // Obtain the response to the SQL request.
    response, err := client.SQLQuery(request)
    if err != nil {
        panic(err)
    }
    // Obtain the schema of the returned results of the SQL request.
    columns := response.ResultSet.Columns()
    fmt.Printf("response table schema: %v\n", columns)
    // Use SQL ResultSet to obtain all returned results of the SQL request.
    fmt.Println("response resultset:")
    resultSet := response.ResultSet
    for resultSet.HasNext() {
        row := resultSet.Next()
        tableName, err := row.GetString(0)
        if err != nil {
            panic(err)
        }
        fmt.Println(tableName)
    }
}
```

Sample output:

```
response table schema: [Tables_in_$instanceName:STRING]
response resultset:
test_table
```

3.10.5. Query information about tables

You can execute the describe statement to query information about tables, such as the field names and field types.

> **Note** For more information about the describe statement, see Query the information about a table.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `describe test_table` statement to query information about the table named test_table.

```
func getTableDesc(client *tablestore.TableStoreClient) {
    // Create a SQL request.
    request := &tablestore.SQLQueryRequest{Query: "describe test_table"}
    // Obtain the response to the SQL request.
    response, err := client.SQLQuery(request)
    if err != nil {
        panic(err)
    }
    // Obtain the schema of the returned results of the SQL request.
    columns := response.ResultSet.Columns()
    fmt.Printf("response table schema: %v\n", columns)
    // Use SQL ResultSet to obtain all returned results of the SQL request.
    fmt.Println("response resultset:")
    resultSet := response.ResultSet
    for resultSet.HasNext() {
        row := resultSet.Next()
        for i := range columns {
            if i > 0 {
                fmt.Printf(", ")
            }
            value, err := row.GetString(i)
            if err != nil {
                panic(err)
            }
            fmt.Printf(value)
        }
        fmt.Println()
    }
}
```

Sample output:

```
response table schema: [Field:STRING Type:STRING Null:STRING Key:STRING Default:STRING Extr
a:STRING]
response resultset:
pk, varchar(1024), NO, PRI, ,
long_value, bigint(20), YES, , ,
double_value, double, YES, , ,
string_value, mediumtext, YES, , ,
bool_value, tinyint(1), YES, , ,
```

3.10.6. Query index information about tables

You can execute the show index statement to query the index information about tables.

> **Note** For more information about the show index statement, see Query the index information about a table.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `show index in test_table` statement to query index information about the table named test_table.

```
func showIndex(client *tablestore.TableStoreClient) {
    // Create a SQL request.
    request := &tablestore.SQLQueryRequest{Query: "show index in test_table"}
    // Obtain the response to the SQL request.
    response, err := client.SQLQuery(request)
    if err != nil {
        panic(err)
    }
    // Obtain the schema of the returned results of the SQL request.
    columns := response.ResultSet.Columns()
    fmt.Printf("response table schema: %v\n", columns)
    // Use SQL ResultSet to obtain all returned results of the SQL request.
    fmt.Println("response resultset:")
    resultSet := response.ResultSet
    for resultSet.HasNext() {
        row := resultSet.Next()
        tableName, _ := row.GetStringByName("Table")
        fmt.Printf("%v, ", tableName)
        nonUnique, _ := row.GetInt64ByName("Non_unique")
        fmt.Printf("%v, ", nonUnique)
        keyName, _ := row.GetStringByName("Key_name")
        fmt.Printf("%v, ", keyName)
        seqInIndex, _ := row.GetInt64ByName("Seq_in_index")
        fmt.Printf("%v, ", seqInIndex)
        columnName, _ := row.GetStringByName("Column_name")
        fmt.Printf("%v, ", columnName)
        indexType, _ := row.GetStringByName("Index_type")
        fmt.Printf("%v\n", indexType)
    }
}
```

Sample output:

```
response table schema: [Table:STRING Non_unique:INTEGER Key_name:STRING Seq_in_index:INTEGE
R Column_name:STRING Is_defined_column:STRING Search_type:STRING Collation:STRING Cardinali
ty:INTEGER Sub_part:INTEGER Packed:STRING Null:STRING Index_type:STRING Comment:STRING Inde
x_comment:STRING Visible:STRING Expression:STRING]
response resultset:
test_table, 0, PRIMARY, 1, pk,
test_table, 1, test_table_index, 1, pk, SearchIndex
test_table, 1, test_table_index, 2, bool_value, SearchIndex
test_table, 1, test_table_index, 3, double_value, SearchIndex
test_table, 1, test_table_index, 4, long_value, SearchIndex
test_table, 1, test_table_index, 5, string_value, SearchIndex
```

3.10.7. Query data

You can execute the select statement to query data in a table.

> **Note** For more information about the select statement, see Query data.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A mapping table is created. For more information, see Create mapping tables for tables.

Parameters

Parameter	Description
query	The SQL statement. Configure the parameter based on the required feature.

Examples

Execute the `select pk, long_value, double_value, string_value, bool_value from test_table limit 20` statement to query data in the table named test_table and set the maximum number of rows that you want to return to 20. The system returns the request type, the schema of the returned results, and the returned results of the query statement.

```
func queryData(client *tablestore.TableStoreClient) {
    // Create a SQL request.
    request := &tablestore.SQLQueryRequest{Query: "select pk, long_value, double_value, str
ing_value, bool_value from test_table limit 20"}
    // Obtain the response to the SQL request.
    response, err := client.SQLQuery(request)
    if err != nil {
        panic(err)
    }
    // Obtain the SQL request type.
    fmt.Printf("response type: %v\n", response.StmtType)
    // Obtain the schema of the returned results of the SQL request.
    columns := response.ResultSet.Columns()
    fmt.Printf("response table meta: %v\n", columns)
    // Obtain the returned results of the SQL request.
    resultSet := response.ResultSet
    fmt.Println("response resultset:")
    for resultSet.HasNext() {
        row := resultSet.Next()
        stringValue, _ := row.GetString(0)
        fmt.Printf("%v, ", stringValue)
        stringValue, _ = row.GetStringByName("pk")
        fmt.Printf("%v, ", stringValue)
        longValue, _ := row.GetFloat64(1)
        fmt.Printf("%v, ", longValue)
        longValue, _ = row.GetFloat64ByName("long_value")
        fmt.Printf("%v, ", longValue)
        floatValue, _ := row.GetFloat64(2)
        fmt.Printf("%v, ", floatValue)
        floatValue, _ = row.GetFloat64ByName("double_value")
        fmt.Printf("%v, ", floatValue)
        stringValue, _ = row.GetString(3)
        fmt.Printf("%v, ", stringValue)
        stringValue, _ = row.GetStringByName("string_value")
        fmt.Printf("%v, ", stringValue)
        boolValue, _ := row.GetBool(4)
        fmt.Printf("%v, ", boolValue)
        boolValue, _ = row.GetBoolByName("bool_value")
        fmt.Printf("%v\n", boolValue)
    }
}
```

Sample output:

```
response type: SQL_SELECT
response table meta: [pk:STRING long_value:INTEGER double_value:DOUBLE string_value:STRING
bool_value:BOOLEAN]
response resultset:
binary_null, binary_null, 0, 0, 1, 1, a, a, false, false
bool_null, bool_null, 0, 0, 1, 1, a, a, false, false
double_null, double_null, 0, 0, 1, 1, a, a, false, false
long_null, long_null, 0, 0, 1, 1, a, a, false, false
string_null, string_null, 0, 0, 1, 1, , , false, false
```

3.11. Data Delivery

3.11.1. Create delivery tasks

You can call the CreateDeliveryTask operation to create a delivery task. You can create a delivery task to deliver data from Tablestore to an OSS bucket.

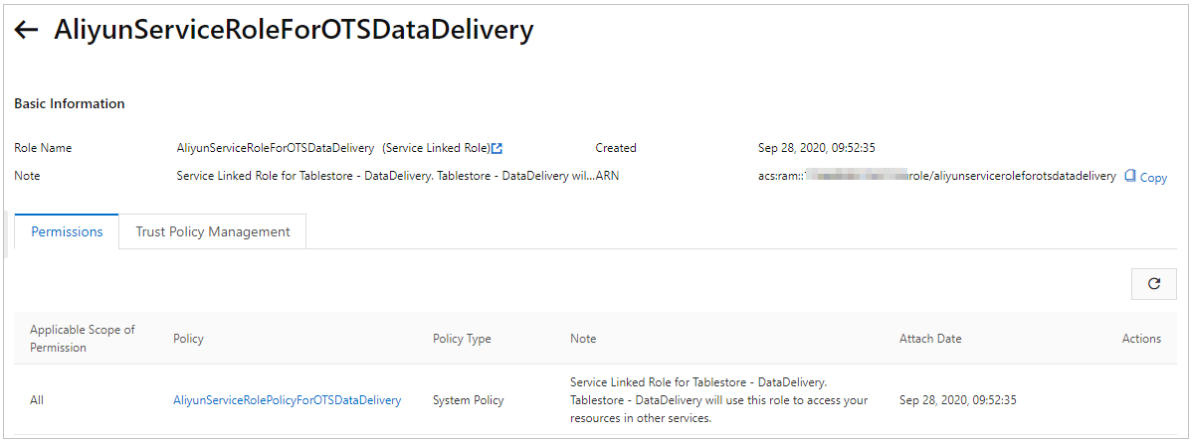
> **Notice**
>
> Make sure that you have installed Tablestore SDK for Go that supports delivery service.

Prerequisites

- OSS is activated. A bucket is created in the region where a Tablestore instance is deployed. For more information, see Activate OSS.
- The Tablestore service linked role (AliyunServiceRoleForOTSDataDelivery) is created in the console. The Alibaba Cloud Resource Name (ARN) of the role is recorded. For more information, see the "Create delivery tasks" section of the Quick start topic.

  You can obtain the ARN of the Tablestore service linked role (AliyunServiceRoleForOTSDataDelivery) in the RAM console. Perform the following operations:

  On the **RAM Roles** page, search AliyunServiceRoleForOTSDataDelivery. Then, click the RAM role name. On the role details page, you can view and copy the ARN information of the role.



- TableStoreClient is initialized. For more information, see Initialization.
- A table is created. Data is written to the table.

Parameters

Parameter	Description
TableName	The name of the table.
TaskName	The name of the delivery task. The name must be 3 to 16 characters in length and can contain only lowercase letters, digits, and hyphens (-). It must start and end with a lowercase letter or digit.

Parameter	Description

TaskConfig	The configurations of the delivery task. Valid values:
	• OssPrefix: the prefix of the folder in the bucket. Data is delivered from Tablestore to the folder. The path of the destination folder supports the following time variables: $yyyy, $MM, $dd, $HH, and $mm.
	  ◦ When the path uses time variables for delivery, OSS folders are dynamically generated based on the time when data is written. This way, data is partitioned based on the naming conventions that are followed when Hive partitions data. Objects in OSS are organized, partitioned, and distributed based on time.
	  ◦ When the path does not use time variables, all files are delivered to an OSS folder whose name contains a specific prefix.
	• OssBucket: the name of the OSS bucket.
	• OssEndpoint: the endpoint of the region where an OSS bucket is deployed.
	• OssRoleName: the ARN of the Tablestore service linked role.
	• Format: the format of the delivered data. The delivered data is stored in the Parquet format. By default, delivery service uses PLAIN to encode data of any type.
	• EventTimeColumn: the event time column. This parameter specifies that data is partitioned based on the time of a column. If you do not specify this parameter, data is partitioned based on the time when the data is written to Tablestore.
	• Schema: specifies the column you want to deliver. You must configure the source fields, destination fields, and destination field types to deliver the column.
	The order in which fields are sorted in Tablestore can be different from the order of fields in the schema. Parquet data stored in OSS is distributed based on the order of fields in the schema.
	Notice
	The data types must be consistent between the source and destination fields. If the data types between the fields are not consistent, the fields are discarded as dirty data. For more information about field type mappings, see the "Data type mappings" section of the Quick start topic.

Parameter	Description
TaskType	The mode in which to deliver data. Default value: BaseIncTask. Valid values: • IncTask: the incremental data delivery mode. Only incremental data is synchronized. • BaseTask: the full data delivery mode. All data in the table is scanned and synchronized. • BaseIncTask: the differential data delivery mode. After the full data is synchronized, Tablestore synchronizes the incremental data. When you synchronize data in incremental mode, you can view the time when data is last delivered and the status of the current delivery task.

Examples

```
func CreateTaskSample(client *tablestore.TableStoreClient) {
 createTask := &tablestore.CreateDeliveryTaskRequest{
  TableName: "sampleTable",
  TaskName: "sampledeliverytask",
  TaskType: tablestore.BaseIncTask,
  TaskConfig: &tablestore.OSSTaskConfig{
   OssPrefix:   "sample/year=$yyyy/month=$MM",
   OssBucket:      "datadeliverytest",
   OssEndpoint:    "oss-cn-hangzhou.aliyuncs.com",
   OssRoleName:    "acs:ram::17************45:role/aliyunserviceroleforotsdatadelivery",
   Schema: []*tablestore.TaskSchema{
    {
     ColumnName: "PK1",
     OssColumnName: "PK1",
     Type: tablestore.ParquetInt64,
    },
    {
     ColumnName: "PK2",
     OssColumnName: "PK2",
     Type: tablestore.ParquetUtf8,
    },
    {
     ColumnName: "Col1",
     OssColumnName: "Col1",
     Type: tablestore.ParquetDouble,
    },


   },
  },
 }
 createResp, err := client.CreateDeliveryTask(createTask)
 if err != nil {
  log.Fatal("create delivery task failed ", err)
 }
 fmt.Println("create delivery task success ", createResp.RequestId)
}
```

3.11.2. List the names of delivery tasks

You can call the ListDeliveryTask operation to list information about all deliver tasks for a table.

Prerequisites

- The TableStoreClient instance is initialized. For more information, see Initialization.
- The delivery task is created. For more information about how to use the console or the SDK, see Quick start or Create delivery tasks.

Request parameters

Parameter	Description
TableName	The name of the table.

Response parameters

Parameter	Description
TableName	The name of the table, which is the same as the name of the table in the request.
TaskName	The name of the delivery task.
TaskType	The mode in which to deliver data. Valid values: • 0: the full data delivery mode • 1: the incremental data delivery mode • 2: the differential data delivery mode

Examples

```
func ListTask(client *tablestore.TableStoreClient, tableName string) {
 resp, err := client.ListDeliveryTask(&tablestore.ListDeliveryTaskRequest{
  TableName: tableName,
 })
 if err != nil {
  log.Fatal("list delivery task failed ", err)
 }
 for _, task := range resp.Tasks {
  fmt.Println("task: ", task)
 }
 fmt.Println("list task finish")
}
```

3.11.3. Query the descriptive information of delivery tasks

You can call the DescribeDeliveryTask operation to query the descriptive information of a delivery task.

Prerequisites

- The TableStoreClient instance is initialized. For more information, see Initialization.

- The delivery task is created. For more information about how to use the console or the SDK, see Quick start or Create delivery tasks.

Request parameters

Parameter	Description
TableName	The name of the table.
TaskName	The name of the delivery task.

Response parameters

Parameter	Description
TaskConfig	The configuration information of the delivery task.
TaskSyncStat	The synchronization status of the delivery task.
TaskType	The mode in which to deliver data. Valid values: • 0: the full data delivery mode • 1: the incremental data delivery mode • 2: the differential data delivery mode

Examples

```
func DescribeTaskSample(client *tablestore.TableStoreClient, tableName, taskName string) {
 req := &tablestore.DescribeDeliveryTaskRequest{
  TableName: tableName,
  TaskName: taskName,
 }
 resp, err := client.DescribeDeliveryTask(req)
 if err != nil {
  log.Fatal("describe delivery task failed ", err)
 }
 fmt.Println("TaskConfig: ", *resp.TaskConfig)
 fmt.Println("TaskSyncStat: ", *resp.TaskSyncStat)
 fmt.Println("TaskType: ", resp.TaskType)
 return
}
```

3.11.4. Delete delivery tasks

You can call the DeleteDeliveryTask operation to delete a delivery task.

Parameters

Parameter	Description
TableName	The name of the table.
TaskName	The name of the delivery task.

Examples

```
func DeleteTaskSample(client *tablestore.TableStoreClient, tableName, taskName string) {
 delResp, err := client.DeleteDeliveryTask(&tablestore.DeleteDeliveryTaskRequest{
  TableName: tableName,
  TaskName: taskName,
 })
 if err != nil {
  log.Fatal("delete delivery task failed ", err)
 }
 fmt.Println("delete task success", delResp.RequestId)
}
```

3.12. TimeSeries model

3.12.1. Overview

Tablestore SDK for Go provides a variety of operations that you can perform on time series tables.

Operation	Description
Create a time series table	Creates a time series table.
Query the names of time series tables	Queries the names of all time series tables in the current instance.
Query the information about a time series table	Queries the information about a time series table, such as the time to live (TTL) configuration.
Update a time series table	Updates the configurations of a time series table.
Delete a time series table	Deletes a time series table.
Write time series data	Writes multiple rows of time series data to a time series table at the same time.
Query time series data	Queries the time series data that meets the specified conditions.

Operation	Description
Retrieve time series	Retrieves time series that meet multiple specified conditions.
Update time series metadata	Updates the properties in the metadata of multiple time series at the same time.

3.12.2. Create a time series table

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

Prerequisites

- An instance is created in the Tablestore console. For more information, see Create an instance for the TimeSeries model.
- The TimeseriesClient is initialized. For more information, see Initialization.

Usage note

The name of the time series table that you create cannot be the same as the name of an existing data table.

Parameters

Example

```
/**
 * Use CreateTimeseriesTableSample to create a time series table named timeseriesTableName
and specify the time to live (TTL) of the data in the time series table.
 */
func CreateTimeseriesTableSample(client *tablestore.TimeseriesClient, timeseriesTableName s
tring , timeToLive int64) {
    fmt.Println("[Info]: Begin to create timeseries table: " , timeseriesTableName)
    timeseriesTableOptions := tablestore.NewTimeseriesTableOptions(timeToLive)        // Co
nstruct the configurations of the time series table.
    // Construct the metadata of the time series table.
    timeseriesTableMeta := tablestore.NewTimeseriesTableMeta(timeseriesTableName)    // Spe
cify the name of the time series table.
    timeseriesTableMeta.SetTimeseriesTableOptions(timeseriesTableOptions)            // Spe
cify the configurations of the time series table.
    createTimeseriesTableRequest := tablestore.NewCreateTimeseriesTableRequest()    // Cons
truct the request to create the time series table.
    createTimeseriesTableRequest.SetTimeseriesTableMeta(timeseriesTableMeta)
    createTimeseriesTableResponse , err := client.CreateTimeseriesTable(createTimeseriesTab
leRequest)    // Call the client to create the time series table.
    if err != nil {
        fmt.Println("[Error]: Failed to create timeseries table with error: " , err)
        return
    }
    fmt.Println("[Info]: CreateTimeseriesTable finished! RequestId: " , createTimeseriesTab
leResponse.RequestId)
}
```

3.12.3. Query the names of time series tables

You can call the ListTimeseriesTable operation to query the names and metadata of all time series tables in the current instance.

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Example

```
/**
 * Use ListTimeseriesTableSample to query the names and metadata of all time series tables
in an instance.
 */
func ListTimeseriesTableSample(client *tablestore.TimeseriesClient) {
    fmt.Println("[Info]: Begin to list timeseries table !")
    listTimeseriesTableResponse , err := client.ListTimeseriesTable()
    if err != nil {
        fmt.Println("[Info]: List timeseries table failed with error: " , err)
    }
    fmt.Println("[Info]: Timeseries table Meta: ")
    for i := 0; i < len(listTimeseriesTableResponse.GetTimeseriesTableMeta()); i++ {
        curTimeseriesTableMeta := listTimeseriesTableResponse.GetTimeseriesTableMeta()[i]
        fmt.Println("[Info]: Timeseries table name: " , curTimeseriesTableMeta.GetTimeserie
sTableName() , "TTL: " , curTimeseriesTableMeta.GetTimeseriesTableOPtions().GetTimeToLive()
)
    }
    fmt.Println("[Info]: ListTimeseriesTableSample finished !")
}
```

3.12.4. Query the information about a time series table

You can call the DescribeTimeseriesTable operation to query the information about a time series table, such as the time to live (TTL) configuration.

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameter

Parameter	Description
timeseriesTableName	The name of the time series table.

Example

```
/**
 * Use DescribeTimeseriesTableSample to query the information about a time series table nam
ed timeseriesTableName.
 */
func DescribeTimeseriesTableSample(client *tablestore.TimeseriesClient , timeseriesTableNam
e string) {
    fmt.Println("[Info]: Begin to require timeseries table description!")
    describeTimeseriesTableRequest := tablestore.NewDescribeTimeseriesTableRequset(timeseri
esTableName)  // Construct the request with the name of the time series table specified.
    describeTimeseriesTableResponse , err := client.DescribeTimeseriesTable(describeTimeser
iesTableRequest)
    if err != nil {
        fmt.Println("[Error]: Failed to require timeseries table description!")
        return
    }
    fmt.Println("[Info]: DescribeTimeseriesTableSample finished. Timeseries table meta: ")
    fmt.Println("[Info]: TimeseriesTableName: " , describeTimeseriesTableResponse.GetTimese
riesTableMeta().GetTimeseriesTableName())
    fmt.Println("[Info]: TimeseriesTable TTL: " , describeTimeseriesTableResponse.GetTimese
riesTableMeta().GetTimeseriesTableOPtions().GetTimeToLive())
}
```

3.12.5. Update a time series table

You can call the UpdateTimeseriesTable operation to update the configurations of a time series table, such as the time to live (TTL).

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

For more information, see Create a time series table.

Example

```
/**
 * Use UpdateTimeseriesTableSample to update the time to live (TTL) configuration of a time
series table.
 */
func UpdateTimeseriesTableSample(client *tablestore.TimeseriesClient , timeseriesTableName
string) {
    fmt.Println("[Info]: Begin to update timeseries table !")
    // Construct the TTL configuration of the time series table.
    timeseriesTableOptions := tablestore.NewTimeseriesTableOptions(964000)
    // Construct the update request.
    updateTimeseriesTableRequest := tablestore.NewUpdateTimeseriesTableRequest(timeseriesTa
bleName)
    updateTimeseriesTableRequest.SetTimeseriesTableOptions(timeseriesTableOptions)
    // Call the time series client to update the time series table.
    updateTimeseriesTableResponse , err := client.UpdateTimeseriesTable(updateTimeseriesTab
leRequest)
    if err != nil {
        fmt.Println("[Error]: Update timeseries table failed with error: " , err)
        return
    }
    DescribeTimeseriesTableSample(client , timeseriesTableName)
    fmt.Println("[Info]: UpdateTimeseriesTableSample finished! RequestId: " , updateTimeser
iesTableResponse.RequestId)
}
```

3.12.6. Delete a time series table

You can call the DeleteTimeseries operation to delete a time series table.

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameter

Parameter	Description
timeseriesTableName	The name of the time series table.

Example

```
/**
 * Use DeleteTimeseriesTableSample to delete a time series table named timeseriesTableName
in the instance.
 */
func DeleteTimeseriesTableSample(client *tablestore.TimeseriesClient , timeseriesTableName
string) {
    fmt.Println("[Info]: Begin to delete timeseries table !")
    // Construct the request to delete the time series table.
    deleteTimeseriesTableRequest := tablestore.NewDeleteTimeseriesTableRequest(timeseriesTa
bleName)
    // Call the time series client to delete the time series table.
    deleteTimeseriesTableResponse , err := client.DeleteTimeseriesTable(deleteTimeseriesTab
leRequest)
    if err != nil {
        fmt.Println("[Error]: Delete timeseries table failed with error: " , err)
        return
    }
    fmt.Println("[Info]: DeleteTimeseriesTableSample finished ! RequestId: " , deleteTimese
riesTableResponse.RequestId)
}
```

3.12.7. Write time series data

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

Prerequisites

- A time series table is created. For more information, see Create a time series table.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

Example

```
/**
 * Use PutTimeseriesDataSample to write one or more rows of time series data to a time seri
es table.
 */
func PutTimeseriesDataSample(client *tablestore.TimeseriesClient , timeseriesTableName stri
ng) {
    fmt.Println("[Info]: Begin to PutTimeseriesDataSample !")
    // Construct timeseriesRow.
    timeseriesKey := tablestore.NewTimeseriesKey()
    timeseriesKey.SetMeasurementName("CPU")
    timeseriesKey.SetDataSource("127.0.0.1")
    timeseriesKey.AddTag("City" , "Hangzhou")
    timeseriesKey.AddTag("Region" , "Xihu")
    timeseriesRow := tablestore.NewTimeseriesRow(timeseriesKey)
    timeseriesRow.SetTimeInus(time.Now().UnixNano() / 1000)
    timeseriesRow.AddField("temperature" , tablestore.NewColumnValue(tablestore.ColumnType_
INTEGER , 98))
```

```
    timeseriesRow.AddField("status" , tablestore.NewColumnValue(tablestore.ColumnType_STRIN
G , "ok"))
    // Construct timeseriesRow1.
    timeseriesKey1 := tablestore.NewTimeseriesKey()
    timeseriesKey1.SetMeasurementName("NETWORK")
    timeseriesKey1.SetDataSource("127.0.0.1")
    timeseriesKey1.AddTag("City" , "Hangzhou")
    timeseriesKey1.AddTag("Region" , "Xihu")
    timeseriesRow1 := tablestore.NewTimeseriesRow(timeseriesKey1)
    timeseriesRow1.SetTimeInus(time.Now().UnixNano() / 1000)
    timeseriesRow1.AddField("in" , tablestore.NewColumnValue(tablestore.ColumnType_INTEGER
, 1000))
    timeseriesRow1.AddField("data" , tablestore.NewColumnValue(tablestore.ColumnType_BINARY
, []byte("tablestore")))
    timeseriesRow1.AddField("program" , tablestore.NewColumnValue(tablestore.ColumnType_STR
ING , "tablestore.d"))
    timeseriesRow1.AddField("status" , tablestore.NewColumnValue(tablestore.ColumnType_BOOL
EAN, true))
    timeseriesRow1.AddField("lossrate" , tablestore.NewColumnValue(tablestore.ColumnType_DO
UBLE , float64(1.9098)))
    // Construct the request to write the time series data.
    putTimeseriesDataRequest := tablestore.NewPutTimeseriesDataRequest(timeseriesTableName)
    putTimeseriesDataRequest.AddTimeseriesRows(timeseriesRow , timeseriesRow1)
    // Call the time series client to write the time series data.
    putTimeseriesDataResponse , err := client.PutTimeseriesData(putTimeseriesDataRequest)
    if err != nil {
        fmt.Println("[Error]: Put timeseries data Failed with error: " , err)
        return
    }
    if len(putTimeseriesDataResponse.GetFailedRowResults()) > 0 {
        fmt.Println("[Warning]: Put timeseries data finished ! Some of timeseries row put F
ailed: ")
        for i := 0; i < len(putTimeseriesDataResponse.GetFailedRowResults()); i++ {
            FailedRow := putTimeseriesDataResponse.GetFailedRowResults()[i]
            fmt.Println("[Warning]: Failed Row: Index: " , FailedRow.Index , " Error: " , F
ailedRow.Error)
        }
    } else {
        fmt.Println("[Info]: PutTimeseriesDataSample finished! RequestId: " , putTimeseries
DataResponse.RequestId)
    }
}
```

3.12.8. Query time series data

You can call the GetTimeseriesData operation to query the time series data that meets the specified conditions in a time series table.

Prerequisites

- Time series data is written to the time series table. For more information, see Write time series data.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

Example

```
/**
 * Use GetTimeseriesDataSample to query the time series data in a specific time series in a
time series table based on timeseriesKey.
 */
func GetTimeseriesDataSample(client *tablestore.TimeseriesClient , timeseriesTableName stri
ng) {
    fmt.Println("[Info]: Begin to get timeseries data !")
    // Construct the timeseriesKey for the time series in which you want to query the time
series data.
    timeseriesKey := tablestore.NewTimeseriesKey()
    timeseriesKey.SetMeasurementName("NETWORK")
    timeseriesKey.SetDataSource("127.0.0.1")
    timeseriesKey.AddTag("City" , "Hangzhou")
    timeseriesKey.AddTag("Region" , "Xihu")
    // Construct the query request.
    getTimeseriesDataRequest := tablestore.NewGetTimeseriesDataRequest(timeseriesTableName)
    getTimeseriesDataRequest.SetTimeseriesKey(timeseriesKey)
    getTimeseriesDataRequest.SetTimeRange(0 , time.Now().UnixNano() / 1000) // Specify the
time range for the query.
    getTimeseriesDataRequest.SetLimit(-1)
    // Call the time series client to query the data in the time series.
    getTimeseriesResp , err := client.GetTimeseriesData(getTimeseriesDataRequest)
    if err != nil {
        fmt.Println("[Error]: Get timeseries data Failed with error: " , err)
        return
    }
    fmt.Println("[Info]: Get timeseries data succeed ! TimeseriesRows: ")
    for i := 0; i < len(getTimeseriesResp.GetRows()); i++ {
        fmt.Println("[Info]: Row" , i , ": [" , getTimeseriesResp.GetRows()[i].GetTimeserie
sKey().GetMeasurementName(),
            getTimeseriesResp.GetRows()[i].GetTimeseriesKey().GetDataSource(),
            getTimeseriesResp.GetRows()[i].GetTimeseriesKey().GetTagsSlice(), "]",
            getTimeseriesResp.GetRows()[i].GetFieldsSlice(),
            getTimeseriesResp.GetRows()[i].GetTimeInus())
    }
    fmt.Println("[Info]: GetTimeseriesDataSample finished! RequestId: " , getTimeseriesResp
.RequestId)
}
```

3.12.9. Retrieve time series

You can call the QueryTimeseriesMeta operation to retrieve time series that meet multiple specified conditions.

Prerequisites

- Time series data is written to the time series table. For more information, see Write time series data.

- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

Example

```
/**
 * Use QueryTimeseriesMetaSample to query the metric name, data source, and tag information of a specific time series in a time series table based on a specific condition, which can be a composite condition.
 */
func QueryTimeseriesMetaSample(client *tablestore.TimeseriesClient , timeseriesTableName string) {
    fmt.Println("[Info]: Begin to query timeseries table meta!")
    // Construct multiple query conditions.
    measurementMetaQueryCondition := tablestore.NewMeasurementQueryCondition(tablestore.OP_GREATER_EQUAL , "")
    datasourceMetaQueryCondition := tablestore.NewDataSourceMetaQueryCondition(tablestore.OP_GREATER_EQUAL , "")
    tagMetaQueryCondition := tablestore.NewTagMetaQueryCondition(tablestore.OP_GREATER_THAN , "City" , "")
    // Construct a composite condition.
    compsiteMetaQueryCondition := tablestore.NewCompositeMetaQueryCondition(tablestore.OP_AND)
    compsiteMetaQueryCondition.AddSubConditions(measurementMetaQueryCondition)
    compsiteMetaQueryCondition.AddSubConditions(datasourceMetaQueryCondition)
    compsiteMetaQueryCondition.AddSubConditions(tagMetaQueryCondition)
    // Construct the query request.
    queryTimeseriesMetaRequest := tablestore.NewQueryTimeseriesMetaRequest(timeseriesTableName)
    queryTimeseriesMetaRequest.SetCondition(compsiteMetaQueryCondition)
    queryTimeseriesMetaRequest.SetLimit(-1)
    // Call the time series client to execute the query request.
    queryTimeseriesTableResponse , err := client.QueryTimeseriesMeta(queryTimeseriesMetaRequest)
    if err != nil {
        fmt.Println("[Error]: Query timeseries table meta failed with error: " , err)
        return
    }
    fmt.Println("[Info]: Query timeseries table meta succeed: ")
    for i := 0; i < len(queryTimeseriesTableResponse.GetTimeseriesMetas()); i++ {
        curTimeseriesMeta := queryTimeseriesTableResponse.GetTimeseriesMetas()[i]
        fmt.Println("[Info]: Meta_" , i , ": " , "Measurement: " , curTimeseriesMeta.GetTimeseriesKey().GetMeasurementName() ,
            "Source: " , curTimeseriesMeta.GetTimeseriesKey().GetDataSource() ,
            "Tags: " , curTimeseriesMeta.GetTimeseriesKey().GetTagsSlice() ,
            "Attrs: " , curTimeseriesMeta.GetAttributeSlice())
    }
    fmt.Println("[Info]: QueryTimeseriesMetaSample finished !")
}
```

3.12.10. Update time series metadata

You can call the UpdateTimeseriesMeta operation to update the metadata of multiple time series at the same time.

Prerequisites

- Time series data is written to the time series table. For more information, see Write time series data.
- The TimeseriesClient is initialized. For more information, see Initialization.

Parameters

The timeseriesMeta parameter specifies the metadata of a time series. Each timeseriesMeta parameter consists of the timeseriesKey and attributes parameters. The following table describes the parameters.

Parameter	Description
timeseriesKey	The identifier of the time series.
attributes	The properties of the time series. The value is a key-value pair of the string type.

Example

```
/**
 * Use UpdateTimeseriesMetaSample to update the properties of time series.
 */
func UpdateTimeseriesMetaSample(tsClient *tablestore.TimeseriesClient, timeseriesTableName
string) {
    fmt.Println("[Info]: Begin to update timeseries meta!")
    PutTimeseriesDataSample(tsClient , timeseriesTableName)
    updateTimeseriesMetaRequest := tablestore.NewUpdateTimeseriesMetaRequest(timeseriesTabl
eName)
    timeseriesKey := tablestore.NewTimeseriesKey()
    timeseriesKey.SetMeasurementName("NETWORK")
    timeseriesKey.SetDataSource("127.0.0.1")
    timeseriesKey.AddTag("City" , "Hangzhou")
    timeseriesKey.AddTag("Region" , "Xihu")
    timeseriesMeta := tablestore.NewTimeseriesMeta(timeseriesKey)
    //timeseriesMeta.SetUpdateTimeInUs(96400)
    timeseriesMeta.AddAttribute("NewRegion" , "Yuhang")
    timeseriesMeta.AddAttribute("NewCity" , "Shanghai")
    updateTimeseriesMetaRequest.AddTimeseriesMetas(timeseriesMeta)
    updateTimeseriesMetaResponse , err := tsClient.UpdateTimeseriesMeta(updateTimeseriesMet
aRequest)
    if err != nil {
        fmt.Println("[Error]: Update timeseries meta failed with error: " , err)
        return
    }
    if len(updateTimeseriesMetaResponse.GetFailedRowResults()) > 0 {
        fmt.Println("[Error]: Update timeseries meta failed row: ")
        for i := 0; i < len(updateTimeseriesMetaResponse.GetFailedRowResults()); i++ {
            fmt.Println("[Error]: " , updateTimeseriesMetaResponse.GetFailedRowResults()[i]
.Index , updateTimeseriesMetaResponse.GetFailedRowResults()[i].Error)
        }
    }
    QueryTimeseriesMetaSample(tsClient , timeseriesTableName)
    fmt.Println("[Info]: UpdateTimeseriesMetaSample finished!")
}
```

4.Python SDK

4.1. Preface

This topic describes how to install and use Tablestore SDK for Python.

Prerequisites

- Tablestore is activated. For more information, see Activate Tablestore.
- An AccessKey pair is created. For more information, see Obtain an AccessKey pair.

Download Tablestore SDK for Python

Download Tablestore SDK for Python from GitHub. For more information, visit GitHub.

For more information about the versions of Tablestore SDK for Python, see Tablestore SDK for Python.

Compatibility

- Tablestore SDK for Python V5.x.x: compatible. Tablestore SDK for Python V5.2.1 is incompatible with Tablestore SDK for Python V5.1.0 in the following conditions:
  - The type of the result returned by the Search operation is different between the two versions.

    By default, Tablestore SDK for Python V5.1.0 and earlier return results of the Tuple type. By default, Tablestore SDK for Python V5.2.0 and later returns SearchResponse objects for Search requests. SearchResponse supports the __iter__ method to traverse data. You can use the SearchResponse.v1_response() method to return results of the Tuple type.
  - The ParallelScan operation is added.

    By default, the results returned for ParallelScan requests are ParallelScanResponse objects. You can use the SearchResponse.v1_response() method to return results of the Tuple type.
- Tablestore SDK for Python V4.x.x: compatible.
- Tablestore SDK for Python V2.x.x: incompatible. Tablestore SDK for Python V2.x.x supports primary key columns in random order, which are no longer supported in Tablestore SDK for Python V4.0.0 and later. Tablestore SDK for Python V4.x.x and Tablestore SDK for Python V2.x.x are incompatible in the following respects:
  - The package name is changed from ots2 to tablestore.
  - The TableOptions parameter is added for the Client.create_table operation.
  - The type of the primary_key parameter is changed from dict to list for the put_row, get_row, and update_row operations to ensure the sequence of primary keys.
  - The type of the attribute_columns parameter is changed from dict to list for the put_row and update_row operations.
  - The timestamp parameter is added to the attribute_columns parameter for the put_row and update_row operations.
  - The max_version and time_range parameters are added for the get_row and get_range operations. You must specify at least one of max_version and time_range.
  - The return_type parameter is added for the put_row, update_row, and delete_row operations. Only RT_PK is supported to return the primary key value of the current row.

- The return_row parameter is added for the put_row, update_row, and delete_row operations. If you set return_type to RT_PK in the request, the primary key value of the row is returned in return_row.

Version

Latest version: 5.2.1.

4.2. Installation

This topic describes how to install Tablestore SDK for Python.

Prerequisites

Python 2 or Python 3 is available.

Install Tablestore SDK for Python

- Method 1: Use pip.

  Run the following command to install Tablestore SDK for Python:

  ```
  sudo pip install tablestore
  ```

- Method 2: Use GitHub.

  Make sure that you have installed Git. For more information about how to download Git, visit Git. After Git is installed, run the following command:

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

- Method 3: Use source code.

  i. Download the SDK package.

  ii. Decompress the SDK package. Then, run the following command:

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

Verify Tablestore SDK for Python

To check the SDK version in the Python environment, enter python on the command line and press Enter.

```
>>> import tablestore
>>> tablestore.__version__
'5.2.1'
```

Uninstall Tablestore SDK for Python

Run the following command to uninstall Tablestore SDK for Python by using pip:

```
sudo pip uninstall tablestore
```

4.3. Initialization

OTSClient is the client for Tablestore. It provides a series of methods for you to manage tables and perform read and write operations on one or more rows.

Obtain an endpoint

An endpoint is a domain name that is used to access a Tablestore instance in a region. The following table describes the examples of endpoints in the supported formats.

Example	Description
http://sun.cn-hangzhou.ots.aliyuncs.com	The public endpoint that is used to access the sun instance in the China (Hangzhou) region over HTTP.
https://sun.cn-hangzhou.ots.aliyuncs.com	The public endpoint that is used to access the sun instance in the China (Hangzhou) region over HTTPS.

Notice You can connect to Tablestore by using public endpoints or private endpoints. For more information, see Endpoint.

To query the endpoints of your Tablestore instance, perform the following steps:

1. Log on to the Tablestore console.
2. Click the name of the instance to go to the **Instance Details** tab.

   On the Instance Details tab, you can view the endpoints of the instance in the Instance Access URL section.

Configure an AccessKey pair

To access Tablestore, you must have an AccessKey pair that consists of an AccessKey ID and AccessKey secret to verify your identity. The following types of AccessKey pairs are supported:

- The AccessKey pair of an Alibaba Cloud account. To obtain the AccessKey pair of an Alibaba Cloud account, take the following steps:
  i. On the Alibaba Cloud official website,Create Your Alibaba Cloud Account.
  ii. Create an AccessKey ID and an AccessKey secret. For more information, see Obtain an AccessKey pair.
- The AccessKey pair of a RAM user who is granted with permissions to access Tablestore. To obtain the AccessKey pair of a RAM user, take the following steps:
  i. Use your Alibaba Cloud account to log on to RAM. Create a RAM user or use an existing RAM user.
  ii. Use your Alibaba Cloud account to authorize the RAM user to access Tablestore.
  iii. After the RAM user is authorized, you can use the AccessKey pair of the RAM user to access Tablestore.
- Temporary access credentials obtained from STS. To obtain temporary access credentials from STS, take the following steps:
  i. The application server uses RAM or STS to obtain access credentials that consist of a temporary AccessKey ID, an AccessKey secret, and a token. After the access credentials are obtained, the application server sends them to you.

ii. You can use the received access credentials to access Tablestore.

Initialize an OTSClient instance

- Operations

```
        """
        Initialize the OTSClient instance.
        end_point indicates the endpoint used to access the Tablestore instance, which mu
st start with https://. Example: https://instance.cn-hangzhou.ots.aliyun.com:80
        access_key_id indicates the AccessKey ID used to access the Tablestore instance.
You can visit the Alibaba Cloud official website or contact an administrator to obtain an
AccessKey ID.
        access_key_secret indicates the AccessKey secret used to access the Tablestore in
stance. You can visit the Alibaba Cloud official website or contact an administrator to o
btain an AccessKey secret.
        instance_name indicates the name of the instance you want to access. You can crea
te an instance in the Tablestore console or contact an administrator to obtain the name o
f an existing instance.
        sts_token indicates the STS token used to access the Tablestore instance. You can
use Alibaba Cloud STS to obtain an STS token. The STS token has a validity period. You mu
st obtain a new token after the existing token expires.
        encoding indicates the method used to encode the request parameter string. The de
fault value is utf8.
        socket_timeout indicates the timeout period in seconds for each socket connection
in the connection pool. It can be an integer or a float number. The default value is 50.
        max_connection indicates the maximum number of connections in a connection pool.
The default value is 50.
        logger_name is used to display the DEBUG log in the request or display the ERROR
log when an error occurs.
        retry_policy defines the retry policy. The default retry policy is DefaultRetryPo
licy. You can define a retry policy based on a RetryPolicy class. For more information, s
ee the code in DefaultRetryPolicy.
        """
        class OTSClient(object):
            def __init__(self, endpoint, access_key_id, access_key_secret, instance_name,
**kwargs):
```

- Examples

```
    #########    Configure the name of the log file and retry policy.     #########
    # The name of the log file is table_store.log. The retry policy is WriteRetryPolicy,
which defines the retry attempts to write the table when a write operation fails.
    ots_client = OTSClient('endpoint', 'access_key_id', 'access_key_secret', 'instance_na
me', logger_name = 'table_store.log',  retry_policy = WriteRetryPolicy())
    #########    Use STS     #########
    ots_client = OTSClient('endpoint', 'STS.K8h*******GB77', 'CkuDj******Wn6', 'instance_
name', sts_token = 'CAISjgJ1q6Ft5B2y********OFcsLLuw==')
```

HTTPS

- HTTPS is supported in Python 2.0.8 and later.

- OpenSSL 0.9.8j or later is required. We recommend that you use OpenSSL 1.0.2d.
- The Python 2.0.8 release package contains a certifi package that can be installed and used. To update the root certificate, download the latest root certificate from Root Certificate.

4.4. Table

4.4.1. 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 search indexes

List search indexes

Query the description of a search index

Delete search indexes

Global secondary index

4.4.2. Create data tables

This topic describes how to create a data table by calling the CreateTable operation. When you call the CreateTable operation, you must specify the schema information and configuration information of the data table when you call the CreateTable operation. You can set reserved read throughput and reserved write throughput for data tables in high-performance instances. You can create one or more index tables when you create a data table.

> Note
>
> - After a data table is created, it takes several seconds to load the data table. During this period, read and write operations performed on the table may fail. Perform operations on the data table after the data table is loaded.
> - You must specify the primary key when you create a data table. A primary key can contain one to four primary key columns. Each primary key column has a name and a data type.

Prerequisites

- An instance is created in the Tablestore console. For more information, see Create instances.
- Client is initialized. For more information, see Initialization.

Operations

```
"""
Description: You can call this operation to create a data table based on the specified tabl
e schema information.
table_meta is an instance of the tablestore.metadata.TableMeta class. table_meta specifies
the name of the data table and the schema of the primary key.
For more information, see the documentation for the TableMeta class. After you create a dat
a table, it takes several seconds to load the partitions in the data table. You can perform
operations on the data table only after the partitions are loaded.
table_options is an instance of the tablestore.metadata.TableOptions class. table_options c
ontains the time_to_live, max_version, and max_time_deviation parameters.
reserved_throughput is an instance of the tablestore.metadata.ReservedThroughput class. res
erved_throughput specifies the reserved read throughput and reserved write throughput.
secondary_indexes is an array that can contain one or more instances of the tablestore.meta
data.SecondaryIndexMeta class. secondary_indexes specifies the global secondary index you w
ant to create.
Return value: none.
"""
def create_table(self, table_meta, table_options, reserved_throughput, secondary_indexes=[]
):
```

Parameters

Parameters	Description

Parameters	Description
table_meta	The schema information of the data table. The schema information contains the following items: • table_name: the name of the data table. • schema_of_primary_key: the schema of the primary key. For more information, see Primary keys and attributes. **Note** Attribute columns are optional. Different rows in Tablestore can have different attribute columns. You can specify the names of attribute columns when you write data to the attribute columns. ◦ The primary key of a data table can contain one to four primary key columns. Primary key columns are sorted in the order in which they are added. For example, PRIMARY KEY (A, B, C) and PRIMARY KEY (A, C, B) have different primary key schemas. Tablestore sorts rows based on the values of all primary key columns. ◦ The first primary key column serves as the partition key. Data that has the same partition key value is stored in the same partition. Therefore, we recommend that you assign no more than 10 GB of data to partitions that have the same partition key value. Otherwise, a single partition may be too large to split. We recommend that you evenly distribute requests to read and write data to partitions that have different partition keys for load balancing. • defined_columns: the predefined columns of the data table and the data types of the predefined column values. Primary key columns cannot be set to predefined columns. You can use predefined columns as the indexed columns or attribute columns for index tables.

Parameters	Description
table_options	The configuration information of the data table. For more information, see Data versions and TTL. The configurations contain the following items: • time_to_live: the period for which data in the table can be retained. This period is the validity period of data. When the retention period exceeds the time_to_live value, Tablestore automatically deletes expired data. The minimum time_to_live value is 86400, which is equal to one day. A value of -1 specifies that data never expires. When you create a data table, you can set time_to_live to -1 if you expect that data never expires. After the data table is created, you can call the UpdateTable operation to modify the time_to_live value. Unit: seconds. **Note** You must set time_to_live to -1 if you want to create an index table for a data table. • max_versions: the maximum number of versions of data that can be retained for a single attribute column. When the number of versions of data in attribute columns exceeds the max_versions value, Tablestore automatically deletes data of earlier versions. When you create a data table, you can customize the maximum number of versions that can be retained for the data in an attribute column. After the data table is created, you can call the UpdateTable operation to modify the max_versions value for the data table. **Note** You must set max_versions to 1 if you want to create an index table for a data table. • max_time_deviation: the max version offset, which is the maximum difference between the timestamp of the written data and the current system time. The difference between the version number and the data written time must be within the value of max_time_deviation. Otherwise, an error occurs when you write the data. Unit: seconds. The valid version range of data in an attribute column is calculated based on the formula: Valid version range = **[Data written time - Max version offset, Data written time + Max version offset)**. When you create a data table, Tablestore uses the default value of 86400 if you do not specify a max version offset. After the table is created, you can call the UpdateTable operation to modify the max_time_deviation value. Unit: seconds.

Parameters	Description
reserved_throughput	The reserved read throughput and reserved write throughput of the data table. The reserved read throughput and reserved write throughput can be set to 0 for data tables only in capacity instances. Reserved throughput does not apply to these instances. The default value 0 specifies that all throughput is billed on a pay-as-you-go basis. Unit: capacity unit (CU). • If reserved read throughput or reserved write throughput is set to a value greater than 0 for a data table, Tablestore allocates and reserves related resources for the data table. After the data table is created, Tablestore charges reserved throughput resources. Additional throughput is billed on a pay-as-you-go basis. For more information, see Billing overview. • If reserved read throughput or reserved write throughput is set to 0 for a data table, Tablestore does not allocate or reserve related resources for the data table.

Parameters	Description
secondary_indexes	The schema information of the index table. The schema information contains the following items: • index_name: the name of the index table. • primary_key_names: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. • defined_column_names: the indexed attribute columns. The attribute columns are a combination of predefined columns of the data table. • index_type: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ◦ If index_type is not specified or set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to index and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ◦ If index_type is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes the data from the indexed columns and primary key columns of a data table to the columns of an index table in synchronous mode. After data is written to the data table, you can query the data from the index table.

Examples

- Create a data table without creating an index table

  The following code provides an example on how to create a data table. The data table contains two primary key columns. In this example, time_to_live is set to 31536000 (one year), max_versions to 3, max_time_deviation to 86400 (one day), and reserved_throughput to (0,0).

```
# Create a schema for primary key columns, including the number, names, and types of the
primary key columns.
# The first primary key column is named pk0 and requires an INTEGER value. The first prim
ary key column is also the partition key.
# The second primary key column is named pk1 and requires an INTEGER value. The data type
is specified as INTEGER in this example. You can also set the data type to STRING or BINA
RY.
schema_of_primary_key = [('pk0', 'INTEGER'), ('pk1', 'INTEGER')]
# Create a tableMeta instance based on the name of the data table and the schema of the p
rimary key columns.
table_meta = TableMeta('SampleTable', schema_of_primary_key)
# Create a TableOptions instance. Set time_to_live to 31536000 to automatically delete ex
pired data, max_versions to 3, and max_time_deviation to 86,400 (one day).
table_options = TableOptions(31536000, 3, 86400)
# Set the reserved read throughput and reserved write throughput to 0.
reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
# Call create_table of the client. If no exceptions occur, the data table is created. If
an exception occurs, the data table fails to be created.
try:
    ots_client.create_table(table_meta, table_options, reserved_throughput)
    print "create table succeeded"
# If the task fails, an exception is returned and handled.
except Exception:
    print "create table failed."
```

For the detailed sample code, visit CreateTable@GitHub.

- Create a data table and an index table whose index type is global secondary index

```
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'], ['bool', 'b', 'd']),
    ]
client.create_table(table_meta, table_option, reserved_throughput, secondary_indexes)
```

- Create a data table and an index table whose index type is local secondary index

```
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', ['gid', 's'], ['bool', 'b', 'd'],index_type= SecondaryIn
dexType.LOCAL_INDEX),
    ]
client.create_table(table_meta, table_option, reserved_throughput, secondary_indexes)
```

4.4.3. UpdateTable

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

Operations

```
    """
        Description: You can call this method to update the attributes of a table. This met
hod can modify only the reserved read/write throughput of a table.
        ``table_name`` specifies the name of the table.
        ``table_options`` is an instance of the ``tablestore.metadata.TableOptions`` class.
It contains the time_to_live, max_version, and max_time_deviation parameters.
        ``reserved_throughput`` is an instance of the ``ots2.metadata.ReservedThroughput``
class. It specifies the reserved read throughput and reserved write throughput.
        Return value: the time when the reserved read throughput or reserved write throughp
ut was raised or lowered most recently for this table and the number of times it was lowere
d today.
        ``update_table_response`` indicates the update result. It is an instance of the ots
2.metadata.UpdateTableResponse class.
        """
        def update_table(self, table_name, table_options, reserved_throughput):
```

Examples

The following code provides an example on how to update the maximum number of versions of a table to five:

```
        # Set both the reserved read throughput and reserved write throughput to 0.
        reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
        # Create a TableOptions instance. Set time_to_live to 31536000 seconds (one year) a
nd delete expired data, max_version to 5, and max_time_deviation to 86400 seconds (one day)
.
        table_options = TableOptions(31536000, 5, 86400)
        try:
            # Call the operation to update the reserved read/write throughput of the table.
            ots_client.update_table('SampleTable', reserved_throughput)
            # If the task succeeds, no exception is returned.
            print "update table succeeded"
        except Exception:
            # If the task fails, an exception is returned for you to handle.
            print "update table failed"
```

For the detailed sample code, visit UpdateTable@GitHub.

4.4.4. List the names of tables

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

> **Note** For more information about the ListTable operation, see ListTable.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created.

API operations

```
"""
Description: You can call this operation to query all table names.
Return value: the table name list.
"table_list" indicates the table name list of the tuple type. Example: ('MyTable1', 'MyTabl
e2').
"""
def list_table(self):
```

Examples

The following code provides an example on how to query 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."
```

For the detailed sample code, visit ListTable@GitHub.

4.4.5. Query the description of a table

> **Note** For more information about the DescribeTable operation, see DescribeTable.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created.

API operations

```
def describe_table(self, table_name):
```

The return value is the description of the table. describe_table_response indicates the description of the table, which is an instance of the ots2.metadata.DescribeTableResponse class.

Parameters

Parameter	Description
table_name	The name of the table.

Examples

The following code provides an example on how to query the description of a table:

```
try:
    describe_response = ots_client.describe_table('myTable')
    # If no exception is thrown, the description is obtained and the following information
is displayed:
    print "describe table succeeded."
    print ('TableName: %s' % describe_response.table_meta.table_name)
    print ('PrimaryKey: %s' % describe_response.table_meta.schema_of_primary_key)
    print ('Reserved read throughput: %s' % describe_response.reserved_throughput_details.c
apacity_unit.read)
    print ('Reserved write throughput: %s' % describe_response.reserved_throughput_details.
capacity_unit.write)
    print ('Last increase throughput time: %s' % describe_response.reserved_throughput_deta
ils.last_increase_time)
    print ('Last decrease throughput time: %s' % describe_response.reserved_throughput_deta
ils.last_decrease_time)
    print ('table options\'s time to live: %s' % describe_response.table_options.time_to_li
ve)
    print ('table options\'s max version: %s' % describe_response.table_options.max_version
)
    print ('table options\'s max_time_deviation: %s' % describe_response.table_options.max_
time_deviation)
except Exception:
    # If an exception is thrown, the operation fails and the system handles the exception.
    print "describe table failed."
```

For the detailed sample code, visit DescribeTable@GitHub.

4.4.6. Delete tables

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

> **Note** For more information about the DeleteTable operation, see DeleteTable.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A table is created.
- The index tables and search indexes for the table are deleted.

Operations

```
def delete_table(self, table_name):
```

Parameters

Parameter	Description
table_name	The name of the table.

Examples

The following code provides an example on how to delete a specified table:

```
try:
    # Call the operation to delete the table named SampleTable.
    ots_client.delete_table('SampleTable')
    # If the task succeeds, no exception is returned.
    print "delete table succeeded"
    # If the task fails, an exception is returned for you to handle.
except Exception:
    print "delete table failed"
```

For the detailed sample code, visit DeleteTable@GitHub.

4.4.7. Configure an auto-increment primary key column

This topic describes how to configure an auto-increment primary key column. You cannot set a partition key to an auto-increment column. If you write data to a table that contains an auto-increment primary key column, you do not need to specify specific values for the auto-increment primary key column because Tablestore generates values for the auto-increment primary key column. Values generated for the auto-increment primary key column are unique and consecutive within the partition that shares the same partition key value.

Prerequisites

The OTSClient instance is initialized. For more information, see Initialization.

Usage notes

1. When you create a table, you cannot set a partition key to an auto-increment primary key column.

   The data type of an auto-increment primary key column can only be set to integer. Each value generated for an auto-increment primary key column is a 64-bit signed long integer.

2. When you write data to a table, you do not need to specify specific values for the auto-increment column. Instead, you need only to set placeholder values for the auto-increment column.

   If you want to obtain the values of the auto-increment column after data is written to the table, you can set ReturnType to RT_PK.

   When you query data, you must specify the values of all primary key columns. To obtain a complete primary key value, you can set ReturnType to RT_PK in PutRow, UpdateRow, or BatchWriteRow.

Examples

When you use auto-increment primary key columns, you can call the CreateTable, PutRow, UpdateRow, or BatchWriteRow operation.

1. Create a table

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

```
from tablestore import *
table_name = 'OTSPkAutoIncrSimpleExample'
def create_table(client):
    # Create a table that includes the following two INTEGER primary keys: gid and uid.
Set uid to an auto-increment column.
    schema_of_primary_key = [('gid', 'INTEGER'), ('uid', 'INTEGER', PK_AUTO_INCR)]
    table_meta = TableMeta(table_name, schema_of_primary_key)
    table_options = TableOptions()
    reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
    client.create_table(table_meta, table_options, reserved_throughput)
    print ('Table has been created.')
```

2. Write data

   When you write data to a table, you do not need to specify specific values for the auto-increment column. Instead, you need only to set placerholder values for the column.

```
from tablestore import *
table_name = 'OTSPkAutoIncrSimpleExample'
def put_row(client):
    # Write data to the primary keys. Set the value of gid to 1 and set uid to the auto
-increment column. You must specify values for the uid column. Otherwise, an error is r
eturned.
    primary_key = [('gid',1), ('uid', PK_AUTO_INCR)]
    attribute_columns = [('name','John'), ('mobile',13900006666), ('address','China'),
('age',20)]
    row = Row(primary_key, attribute_columns)
    # Write data to attribute columns.
    row.attribute_columns = [('name','John'), ('mobile',13900006666), ('address','China
'), ('age',25)]
    consumed, return_row = client.put_row(table_name, row)
    print ('Write succeed, consume %s write cu.' % consumed.write)
    consumed, return_row = client.put_row(table_name, row, return_type = ReturnType.RT_
PK)
    print ('Write succeed, consume %s write cu.' % consumed.write)
    print ('Primary key:%s' % return_row.primary_key)
```

4.4.8. Configure conditional update

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

When you use the PutRow, UpdateRow, DeleteRow, or BatchWriteRow operation to update data, you can use conditional update to check row existence conditions and column-based conditions. The update is successful only when the conditions are met.

Conditional update includes row existence conditions and column-based conditions.

- Row existence conditions: include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.

  When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported.

- Column-based conditions: include SingleColumnCondition and CompositeColumnCondition, which are used to determine whether the conditions are met based on the values of one or more columns.
  - SingleColumnCondition supports the comparison between a constant and a column that can be a primary key column. SingleColumnCondition does not support the comparison between two columns or two constants.
  - The inner node of CompositeColumnCondition is used to perform logical operations. Subconditions can be SingleColumnCondition or CompositeColumnCondition.

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

Parameters

Parameter	Description
RowExistenceExpectation	When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails and an error is reported. Row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST. In Tablestore, RowExistenceExpectation_IGNORE indicates IGNORE, RowExistenceExpectation_EXPECT_EXIST indicates EXPECT_EXIST, and RowExistenceExpectation_EXPECT_NOT_EXIST indicates EXPECT_NOT_EXIST. • IGNORE: No existence check is performed. • EXPECT_EXIST: The row is expected to exist. If the row exists, the condition is met. If the row does not exist, the condition is not met. • EXPECT_NOT_EXIST: Expect that the row does not exist. If the row does not exist, the condition is met. If the row exists, the condition is not met.
column_name	The column name.
column_value	The comparison value of the column.

Parameter	Description
comparator	The relational operator that is used to compare column values. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=). In Tablestore, CT_EQUAL indicates EQUAL (=), CT_NOT_EQUAL indicates NOT_EQUAL (!=), CT_GREATER_THAN indicates GREATER_THAN (>), CT_GREATER_EQUAL indicates GREATER_EQUAL (>=), CT_LESS_THAN indicates LESS_THAN (<), and CT_LESS_EQUAL indicates LESS_EQUAL (<=).
combinator	The logical operator that is used to combine multiple conditions. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. In Tablestore, LO_NOT indicates NOT, LO_AND indicates AND, and LO_OR indicates OR. Different logical operators require different number of subconditions. • When the logical operator is NOT, only one subcondition can be added. • When the logical operator is AND or OR, you must add at least two subconditions.
pass_if_missing	Specifies whether to pass the conditional check when a column does not exist in a row. The type of this parameter value is Boolean. The default value is True, which indicates that if the column does not exist in a row, the conditional check is passed, and the row meets the update conditions. When pass_if_missing is set to False, if the column does not exist in a row, the conditional check fails, and the row does not meet the update conditions.
latest_version_only	Specifies whether to use only the value of the latest version when the comparison value has multiple versions. The type of this parameter value is Boolean. The default value is True, which indicates that if the column has multiple versions of data, only the value of the latest version is used for comparison. When latest_version_only is set to False, the comparison values of all versions are used for comparison if the comparison value has multiple versions. In this case, if only the value of a version meets the condition, the conditional check is passed, and the row meets the update conditions.

Examples

Update a row based on the specified primary key. If the specified row exists and value of the age column in the specified row is 20, the update succeeds. Otherwise, the update fails.

```
    primary_key = [('gid',1), ('uid',"101")]
    update_of_attribute_columns = {
        'PUT' : [('name','David'), ('address','Hongkong')],
        'DELETE' : [('address', None, 1488436949003)],
        'DELETE_ALL' : [('mobile'), ('age')],
        'INCREMENT' : [('counter', -1)]
    }
    row = Row(primary_key, update_of_attribute_columns)
    # Specify that the update succeeds only when the following two conditions are met. Othe
rwise, the update fails.
    # 1 The specified row exists.
    # 2 The value of the age column in the specified row is 20.
    condition = Condition(RowExistenceExpectation.EXPECT_EXIST, SingleColumnCondition("age"
, 20, ComparatorType.EQUAL)) # Update the row only when this row exists.
    consumed, return_row = client.update_row(table_name, row, condition)
```

4.4.9. Configure local transaction

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

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

You can use local transaction to specify that the operations on a partition key are atomic, which indicates that operations based on the specified partition key either succeed or fail. The isolation level of the local transaction is read committed.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

1. Use start_local_transaction to create a local transaction based on the specified partition key value and obtain the ID of the local transaction.
2. Read and write data within the local transaction.

   You can call the following operations on the local transaction: GetRow, PutRow, DeleteRow, UpdateRow, BatchWriteRow, and GetRange.
3. Use commit_transaction to commit the local transaction or use abort_transaction to abort the local transaction.

Limits

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

  If a transaction is not committed or aborted within 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.
- A transaction may be created on the Tablestore server even if a timeout error is returned. In this case,

you can resend a transaction creation request after the created transaction times out.

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

    If a transaction is not read or written for more than 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.
  - Up to 4 MB of data can be written to each transaction. The volume of data written to each transaction is calculated in the same way as a regular write request.
  - If you do not specify a version number for a cell, the Tablestore server assigns a version number to the cell in the usual way when the cell is written to the transaction (rather than when the transaction is committed).
  - If a BatchWriteRow request includes a transaction ID, all rows in the request can be written only to the table that matches the transaction ID.
  - When you use a transaction, the data within the range specified based on the corresponding partition key value is locked. If a request that is sent to write data within the local transaction does not contain the transaction ID, the request fails. Data within the transaction is unlocked when the transaction is committed or aborted, or when the transaction times out.
  - A transaction remains valid even if a read or write request with the transaction ID is rejected. You can resend the request in the same manner as a regular request or you can abort the transaction.

Parameters

Parameter	Description
table_name	The name of the table.
key	The partition key of the table. When you create a local transaction, you need only to specify the partition key value for the local transaction.
primary_key	The primary key of the table. After a local transaction is created, you must specify a complete primary key when you read data from or write data to the local transaction.
transaction_id	The ID that identifies a local transaction. After a local transaction is created, the transaction ID must be included for operations on the local transaction.

Examples

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

```
# Create a local transaction for the partition key PK0.
key = [('PK0', 1)]
# The returned value of the start_local_transaction method is the transaction ID.
transaction_id = client.start_local_transaction(table_name, key)
```

2. Read and write data within the local transaction.

   Local transaction IDs are used to read and write data within the local transaction, which is the same as the normal read and write operations on data.

   - Write a row of data.

```
primary_key = [('PK0', 1), ('PK1', 'transaction')]
attribute_columns = [('value', 'origion value')]
row = Row(primary_key, attribute_columns)
condition = Condition(RowExistenceExpectation.IGNORE)
consumed, return_row = client.put_row(table_name, row, condition)
```

   - Read the row of data.

```
primary_key = [('PK0', 1), ('PK1', 'transaction')]
columns_to_get = ['value']
consumed, return_row, next_token = client.get_row(
    table_name, primary_key, columns_to_get, None, 1, None, None, None, None, transac
tion_id
)
for att in return_row.attribute_columns:
    print ('\tname:%s\tvalue:%s' % (att[0], att[1]))
```

3. Commit or abort a local transaction.

   - Commit a local transaction for all modified data to take effect.

```
client.commit_transaction(transaction_id)
```

   - Abort a local transaction. Modifications to the data within the local transaction are ignored.

```
client.abort_transaction(transaction_id)
```

4.4.10. Configure atomic counter

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

> **Note** Tablestore SDK for Python V5.1.0 and later support the atomic counter feature.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Limits

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

API operations

The API operations to implement the atomic counter feature are added to the updateRow operation. The following table lists these operations.

Operation	Description
update_of_attribute_columns	Increases or decreases a value by a number. The update type is INCREMENT.

Parameters

Parameter	Description
table_name	The name of the table.
column_name	The name of the column on which operations are performed to implement atomic counter. Columns whose valid values are integers are supported to implement the atomic counter feature.
value	The value to update the columns on which operations are performed to implement atomic counter.

Examples

The following code provides an example on how to call the updateRow operation to update the columns whose valid values are integers when you write a row of data. Set the type to INCREMENT for the attribute column:

```
def increment_by_update_row(client):
    primary_key = [('pk0', 1)]
    # The type of the attribute column is INCREMENT. Specify that the value of the price co
lumn is increased by 6.
    update_of_attribute_columns = {
        'INCREMENT': [('price', 6)]
    }
    row = Row(primary_key, update_of_attribute_columns)
    consumed, return_row = client.update_row(table_name, row, None)
    print ('Update succeed, consume %s write cu.' % consumed.write)
```

4.4.11. Filter

Tablestore filters query results on the server side before returning the results. Only rows of data that match the filter conditions are returned.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A table is created. Data is written to the table.

Usage notes

When you call the GetRow, BatchGetRow, or GetRange operation to query data, you can use a filter to return only the rows that meet the filter conditions.

Filters include SingleColumnCondition and CompositeColumnCondition.

- SingleColumnCondition: determines whether to filter a row based only on the values of a reference column.
- CompositeColumnCondition: determines whether to filter a row based on a combination of the filter conditions on values of multiple reference columns.

Limits

- A filter can consist of up to 10 conditions. Supported conditions include relational operator-based operations (=, !=, >, >=, <, and <=) and logical operations (NOT, AND, and OR).
- The reference columns used by a filter must be included in the read data. If the specified columns from which data is read do not include reference columns, the filter cannot query the values of reference columns.
- You can use filters by calling the GetRow, BatchGetRow, and GetRange operations, which does not change the native semantics or limited items of these operations.

  When you use GetRange, the number of rows scanned cannot exceed 5,000, or the data scanned cannot exceed 4 MB in size.

  If the scanned 5,000 rows or 4 MB of data does not match the filter conditions, the rows in the response are empty. However, next_start_primary_key may not be empty. In this case, you must use next_start_primary_key to continue reading the data until next_start_primary_key is empty.

Parameters

Parameter	Description
column_name	The name of the reference column used by the filter.
column_value	The value of the reference column used by the filter.
ComparatorType	The relational operator in the filter. For more information, see ComparatorType. The relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=).
LogicOperator	The logical operator in the filter. For more information, see LogicalOperator. The logical operators include NOT, AND, and OR.
pass_if_missing	Specifies whether to return a row if the row does not contain the reference column. The data type of this parameter value is Boolean. The default value is True, which indicates that if the reference column does not exist in a row, the row is returned. When pass_if_missing is set to False, if the reference column does not exist in a row, the row is not returned.
latest_version_only	Specifies whether to use only the value of the latest version for comparison when the reference column contains data of multiple versions. The data type of this parameter value is Boolean. The default value is True, which indicates that if the reference column has multiple versions of data, only the value of the latest version is used for comparison. When latest_version_only is set to False, if a reference column has multiple versions of data, the values of all versions in the column are used for comparison. In this case, when at least one version of data meets the conditions, the row is returned.

Examples

- The following code provides an example on how to create SingleColumnCondition:

```
def get_row_with_condition(client):
    primary_key = [('uid',1), ('gid',101)]
    columns_to_get = [] # Set the columns to return. If this parameter is not set, all co
lumns are returned.
    // Specify that a row is returned when the value of name is Hangzhou.
    cond = SingleColumnCondition("name", 'Hangzhou', ComparatorType.EQUAL, pass_if_missin
g = True)
    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 ('name:%s\tvalue:%s\ttimestamp:%d' % (att[0], att[1], att[2]))
```

- The following code provides an example on how to create CompositeColumnCondition:

```
def get_row_with_composite_condition(client):
    primary_key = [('uid',1), ('gid',101)]
    columns_to_get = [] # given a list of columns to get, or empty list if you want to ge
t entire row.
    // Specify that a row returned when the value of growth is 0.9, and the value of name
is Hangzhou.
    cond = CompositeColumnCondition(LogicalOperator.AND)
    cond.add_sub_condition(SingleColumnCondition("growth", 0.9, ComparatorType.EQUAL))
    cond.add_sub_condition(SingleColumnCondition("name", 'Hangzhou', ComparatorType.EQUAL
))
    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 ('name:%s\tvalue:%s\ttimestamp:%d' % (att[0], att[1], att[2]))
```

4.5. Single-row operations

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

PutRow

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

- Operations

```
"""
Description: This operation writes a single row of data. Thu number of capacity units (CU
s) consumed by the operation is returned.
table_name specifies the name of the table.
row specifies the row you want to insert to the table, including primary key columns and
attribute columns.
condition is an instance of the tablestore.metadata.Condition class. After conditions are
specified, Tablestore checks whether the specified conditions are met before Tablestore p
erforms the operation. The operation is performed only when the conditions are met. This
feature supports row existence and column-based conditions checks. Check conditions for r
ow existence include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.
return_type specifies the type of data to return. return_type is an instance of the table
store.metadata.ReturnType class. Only primary key columns can be returned. This parameter
applies to auto-increment primary key columns.
Response: the number of capacity units (CUs) consumed by this operation and the row data
to return.
consumed is an instance of the tablestore.metadata.CapacityUnit class. This parameter ind
icates the number of CUs consumed by this operation.
return_row indicates the row to return, including the primary key and attribute columns.
"""
def put_row(self, table_name, row, condition = None, return_type = None)
```

- Parameters

Parameter	Description
table_name	The name of the table.
primary_key	The primary key of the row. **Note** ◦ The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table. ◦ For an auto-increment primary key column, you need only to set the value of the auto-increment primary key column to a placeholder. For more information, see Configure an auto-increment primary key column.
attribute_columns	The attribute columns of the row. ◦ An attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType, which is optional), and timestamp (optional). ◦ You can set ColumnType to ColumnType.INTEGER, ColumnType.STRING, ColumnType.BINARY, ColumnType.BOOLEAN, or ColumnType.DOUBLE, which separately indicates INTEGER, STRING (a UTF-8 encoded string), BINARY, BOOLEAN, or DOUBLE. If you want to set the column value type to BINARY, you must set ColumnType to ColumnType.BINARY. If you want to use other types of column values, the setting of ColumnType is optional. ◦ A timestamp is the data version number. For more information, see Data versions and TTL. You can customize a data version number or use the data version number generated by Tablestore. If you do not specify this parameter, the data version number generated by the system is used. ■ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ■ When you choose to specify the version number, ensure that the version number is a 64-bit timestamp accurate to the millisecond within the valid version range.

Parameter	Description
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update. **Note** ◦ RowExistenceExpectation.IGNORE indicates that new data is inserted into a row regardless of whether the specified row exists or not. If the specified row exists, the existing data is overwritten. ◦ RowExistenceExpectation.EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten. ◦ RowExistenceExpectation.EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist.

- Examples

  The following code provides an example on how to insert a row of data:

  Note In the following example, the data version in the age attribute column is 1498184687000, which indicates June 23, 2017. If **the difference between the current time and max_timed_deviation** (specified during table creation) is greater than 1498184687000, PutRow is disabled.

```
## The first primary key column is gid and its value is the integer 1. The second primary
key column is uid and its value is the integer 101.
primary_key = [('gid',1), ('uid',101)]
## Insert the following attribute columns:
##             The name of the first attribute column is name and its value is John (typ
e: String). The version number is not specified, so the current system time is used as th
e version number.
##             The name of the second attribute column is mobile and its value is 139000
06666 (type: Integer). The version number is not specified, so the current system time is
used as the version number.
##             The name of the third attribute column is address and its value is China
(type: Binary string). The version number is not specified, so the current system time is
used as the version number.
##             The name of the fourth attribute column is female and its value is False
(type: Boolean). The version number is not specified, so the current system time is used
as the version number.
##             The name of the fifth attribute column is age and its value is 29.7. The
specified version number is 1498184687000.
attribute_columns = [('name','John'), ('mobile',13900006666),('address', bytearray('China
')),('female', False), ('age', 29.7, 1498184687000)]
## Construct a row by using primary_key and attribute_columns.
row = Row(primary_key, attribute_columns)
# Set a row existence condition that expects the specified row does not exist. If the spe
cified row exists, the Condition Update Failed error occurs.
condition = Condition(RowExistenceExpectation.EXPECT_NOT_EXIST)
try :
    # If ReturnType is not specified when you call the put_row method, the value of retur
n_row is None.
    consumed, return_row = client.put_row(table_name, row, condition)
    # The write CUs consumed by this request are displayed.
    print ('put row succeed, consume %s write cu.' % consumed.write)
# Client exceptions are caused by parameter errors or network exceptions.
except OTSClientError as e:
    print "put row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.ge
t_error_message())
# Server exceptions are caused by parameter or throttling errors.
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())
```

For the detailed sample code, visit Put Row@Git Hub.

GetRow

You can call this operation to read a row of data.

The following results of the read request may be returned:

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

```
"""
Description: This operation reads a single row of data.
table_name specifies the name of the table.
primary_key specifies the primary key. Type: list.
columns_to_get is optional. This parameter specifies the columns to read. If you do not s
pecify this parameter, all columns are read. Type: list.
column_filter is optional. This parameter specifies the filter conditions for columns. On
ly rows that meet the conditions are returned.
max_version is optional. This parameter specifies the maximum number of versions that can
be read. You must specify at least one of max_version and time_range.
time_range is optional. This parameter specifies a range of versions to read or a version
of data to read. You must specify at least one of max_version and time_range.
Response: the number of CUs consumed by this operation and the primary key columns and at
tribute columns.
consumed is an instance of the tablestore.metadata.CapacityUnit class. This parameter ind
icates the number of CUs consumed by this operation.
return_row indicates the row to return, including the primary key and attribute columns.
Type: list. Example: [('PK0',value0), ('PK1',value1)].
next_token indicates the start column of a wide row to read next time. The data type of t
he column is BINARY.
"""
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):
```

- Parameters

Parameter	Description
table_name	The name of the table.
primary_key	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.

Parameter	Description
columns_to_get	The names of the columns to return. You can specify the names of the primary key columns and the names of attribute columns. If you do not specify this parameter, all data in the row is returned. **Note** ◦ By default, Tablestore returns the data from all columns of the row when you query a row. You can set the columns_to_get parameter to return the data in specified columns. For example, if you include col0 and col1 in columns_to_get, only the values of col0 and col1 are returned. ◦ When columns_to_get and column_filter are used at the same time, the columns specified by columns_to_get are returned. Then, the returned columns are filtered.
max_version	The maximum number of versions that can be read. **Note** You must specify at least one of max_version and time_range. ◦ If you specify only max_version, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, data whose version number is within the specified range or a specified version of data is returned. ◦ If you specify both max_version and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
time_range	Specifies a range of versions to read or a version of data to read. For more information, see TimeRange. **Note** You must specify at least one of max_version and time_range. ◦ If you specify only max_version, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, data whose version number is within the specified range or a specified version of data is returned. ◦ If you specify both max_version and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ If you specify a range of versions to read, you must set start_time and end_time. start_time indicates the start timestamp. end_time indicates the end timestamp. The specified range includes the start timestamp and excludes the end timestamp. ◦ If you want to query data of a specified version, you must set specific_time. specific_time indicates a specified timestamp. You can set either of specific_time or [start_time, end_time). The timestamp used for the value of time_range ranges from 0 to INT64.MAX. Unit: milliseconds.
column_filter	You can set filter conditions to filter the queried results on the server side. Only rows that meet the specified filter conditions are returned. For more information, see Filter. **Note** When columns_to_get and column_filter are used at the same time, the columns specified by columns_to_get are returned. Then, the returned columns are filtered.

- Examples

  The following code provides an example on how to read a row of data:

```
# The first primary key column is uid and its value is the integer 1. The second primary
key column is gid and its value is the integer 101.
primary_key = [('uid',1), ('gid',101)]
# The attribute columns to return are name, growth, and type. If the value of columns_to_
get is [], all attribute columns are returned.
columns_to_get = ['name', 'growth', 'type']
# Set filters for the columns. If the value in the growth column is not 0.9 and the value
in the name column is Hangzhou, this row is returned.
cond = CompositeColumnCondition(LogicalOperator.AND)
cond.add_sub_condition(SingleColumnCondition("growth", 0.9, ComparatorType.NOT_EQUAL))
cond.add_sub_condition(SingleColumnCondition("name", 'Hangzhou', ComparatorType.EQUAL))
try:
    # If you set the value of the last parameter to 1 when you call the get_row operation
, only a version of data is returned in the response.
    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:
        # Display the key, value, and version of each column.
        print ('name:%s\tvalue:%s\ttimestamp:%d' % (att[0], att[1], att[2]))
# Client exceptions are caused by parameter errors or network exceptions.
except OTSClientError as e:
    print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.ge
t_error_message())
# Server exceptions are caused by parameter or throttling errors.
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())
```

For the detailed sample code, visit GetRow@GitHub.

UpdateRow

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

> **Note** If the UpdateRow request contains only columns to delete, and the specified row does not exist, a new row is not added.

- Operations

```
"""
Description: This operation updates a single row of data.
table_name specifies the name of the table.
row specifies the row you want to update, including primary key columns and attribute col
umns of the list type.
condition is an instance of the tablestore.metadata.Condition class. After conditions are
specified, Tablestore checks whether the specified conditions are met before Tablestore p
erforms the operation. The operation is performed only when the conditions are met. This
feature supports row existence and column-based conditions checks. Check conditions for r
ow existence include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.
return_type specifies the type of data to return. return_type is an instance of the table
store.metadata.ReturnType class. Only primary key columns can be returned. This parameter
applies to auto-increment primary key columns.
Response: the number of CUs consumed by this operation and the row data that is specified
by return_row to return.
consumed is an instance of the tablestore.metadata.CapacityUnit class. This parameter ind
icates the number of CUs consumed by this operation.
return_row indicates the row to return.
"""
def update_row(self, table_name, row, condition, return_type = None)
```

- Parameters

Parameter	Description
table_name	The name of the table.
primary_key	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
update_of_attribute_columns	The attribute column to be updated.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.

- Examples

  The following code provides an example on how to update the data of a specified row:

```
# The first primary key column is uid and its value is the integer 1. The second primary
key column is gid and its value is the integer 101.
primary_key = [('uid',1), ('gid',101)]
# The update types include PUT, DELETE, and DELETE_ALL.
# PUT: adds a value to the column or updates the existing value in the column. In the exa
mple, two columns are added. The first column is name and its value is David. The second
column is address and its value is Hongkong.
# DELETE: deletes a specified version (timestamp) of data. In this example, data whose ve
rsion number is 1488436949003 in the address column is deleted.
# DELETE_ALL: deletes the column. In the example, the date of all versions in the mobile
and age columns are deleted.
update_of_attribute_columns = {
    'PUT' : [('name','David'), ('address','Hongkong')],
    'DELETE' : [('address', None, 1488436949003)],
    'DELETE_ALL' : [('mobile'), ('age')],
}
row = Row(primary_key, update_of_attribute_columns)
# The condition for row check is that the row existence is ignored. No matter whether the
row exists, the row is updated.
condition = Condition(RowExistenceExpectation.IGNORE, SingleColumnCondition("age", 20, Co
mparatorType.EQUAL)) # update row only when this row is exist
try:
    consumed, return_row = client.update_row(table_name, row, condition)
# Client exceptions are caused by parameter errors or network exceptions.
except OTSClientError as e:
    print "update row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e
.get_error_message())
# Server exceptions are caused by parameter or throttling errors.
except OTSServiceError as e:
    print "update 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(
))
```

For the detailed sample code, visit UpdateRow@GitHub.

DeleteRow

You can call this operation to delete a row. If the specified row does not exist, no change is made to the table.

- Operations

```
"""
Description: This operation deletes a single row of data.
table_name specifies the name of the table.
row specifies the row you want to delete, including only the primary key.
condition is an instance of the tablestore.metadata.Condition class. After conditions are
specified, Tablestore checks whether the specified conditions are met before Tablestore p
erforms the operation. The operation is performed only when the conditions are met. This
function supports row existence and column-based conditions checks. Check conditions for
row existence include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.
Response: the number of CUs consumed by this operation and the row data that is specified
by return_row to return.
consumed is an instance of the tablestore.metadata.CapacityUnit class. This parameter ind
icates the number of CUs consumed by this operation.
return_row indicates the row to return.
"""
def delete_row(self, table_name, row, condition, return_type = None):
```

- Parameters

Parameter	Description
table_name	The name of the table.
primary_key	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.

- Examples

  The following code provides an example on how to delete a row from a table:

```
primary_key = [('gid',1), ('uid','101')]
row = Row(primary_key)
try:
    consumed, return_row = client.delete_row(table_name, row, None)
# Client exceptions are caused by parameter errors or network exceptions.
except OTSClientError as e:
    print "update row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e
.get_error_message())
# Server exceptions are caused by parameter or throttling errors.
except OTSServiceError as e:
    print "update 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(
))
print ('Delete succeed, consume %s write cu.' % consumed.write)
```

For the detailed sample code, visit DeleteRow@GitHub.

4.6. Multi-row operations

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

BatchWriteRow

You can call this operation to write several rows of data to one or more tables by sending one request. The BatchWriteRow operation consists of PutRow, UpdateRow, and DeleteRow operations. The process of constructing a suboperation is the same as that of calling the PutRow, UpdateRow, and DeleteRow operations. BatchWriteRow supports conditional updates.

The execution and returning of results of each suboperation are carried out independently.

When multiple rows are written, part of rows may fail to be written. If the operation fails for part of rows, the system does not return exceptions. However, information about the index and error messages of the failed rows is contained in BatchWriteRowResponse. Therefore, when you call the BatchWriteRow operation, you must check the return values and check whether the status of each row is successful. If you do not check the return values, failures of operations on part of rows are ignored.

When the server detects errors caused by operations, the BatchWriteRow operation may return exceptions of parameter errors. All operations for the request are not executed.

- Operations

```
"""
Description: This operation writes data to multiple rows.
request = MiltiTableInBatchWriteRowItem()
request.add(TableInBatchWriteRowItem(table0, row_items))
request.add(TableInBatchWriteRowItem(table1, row_items))
response = client.batch_write_row(request)
response indicates the returned result. Type: tablestore.metadata.BatchWriteRowResponse.
"""
def batch_write_row(self, request):
```

- Parameters

  For more information about parameters, see Single-row operations.

- Examples

  The following code provides an example on how to write multiple rows of data:

```
put_row_items = []
# Add rows for PutRow.
for i in range(0, 10):
    primary_key = [('gid',i), ('uid',i+1)]
    attribute_columns = [('name','somebody'+str(i)), ('address','somewhere'+str(i)), ('ag
e',i)]
    row = Row(primary_key, attribute_columns)
    condition = Condition(RowExistenceExpectation.IGNORE)
    item = PutRowItem(row, condition)
    put_row_items.append(item)
# Add rows for 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 for 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 request to write multiple rows.
request = BatchWriteRowRequest()
request.add(TableInBatchWriteRowItem(table_name, put_row_items))
request.add(TableInBatchWriteRowItem('notExistTable', delete_row_items))
# When you call the batch_write_row method to write multiple rows, exceptions may occur i
f errors such as request parameter errors occur. If the operation fails for part of rows,
an exception may not occur but the internal items may fail.
try:
    result = client.batch_write_row(request)
    print ('Result status: %s'%(result.is_all_succeed()))
    # Check the result of inserting rows.
    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.e
rror_message))
    # Check the result of updating rows.
    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, ite
m.error_message))
    # Check the result of deleting rows.
    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, ite
m.error_message))
# Client exceptions are caused by parameter errors or network exceptions.
except OTSClientError as e:
    print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.ge
t_error_message())
# Server exceptions are caused by parameter or throttling errors.
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())
```

For the detailed sample code, visit BatchWriteRow@GitHub.

BatchGetRow

You can call this operation to read several rows of data from one or more tables by sending one request. The BatchGetRow operation consists of multiple GetRow operations. The process of constructng a suboperation is the same as that of calling the GetRow operation. BatchGetRow supports filters.

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

The execution and returning of results of each suboperation are carried out independently.

When multiple rows are read, part of rows may fail to be read. If the operation fails for part of rows, the system does not return exceptions. However, information about the error messages of the failed rows is contained in BatchGetRowResponse. Therefore, when you call the BatchGetRow operation, you must check the return values and check whether the status of each row is successful.

- Operations

```
"""
Description: This operation reads the specified rows of data.
request = BatchGetRowRequest()
request.add(TableInBatchGetRowItem(myTable0, primary_keys, column_to_get=None, column_fil
ter=None))
request.add(TableInBatchGetRowItem(myTable1, primary_keys, column_to_get=None, column_fil
ter=None))
request.add(TableInBatchGetRowItem(myTable2, primary_keys, column_to_get=None, column_fil
ter=None))
request.add(TableInBatchGetRowItem(myTable3, primary_keys, column_to_get=None, column_fil
ter=None))
response = client.batch_get_row(request)
response: the response to the request. Type: tablestore.metadata.BatchGetRowResponse.
"""
def batch_get_row(self, request):
```

- Parameters

  For information about parameters, see Single-row operations.

- Examples

  The following code provides an example on how to read three rows of data at a time:

```
# Specify the columns you want to return.
columns_to_get = ['name', 'mobile', 'address', 'age']
# Read three rows of data.
rows_to_get = []
for i in range(0, 3):
    primary_key = [('gid',i), ('uid',i+1)]
    rows_to_get.append(primary_key)
# Configure filter conditions. Set name to John and address to China.
cond = CompositeColumnCondition(LogicalOperator.AND)
cond.add_sub_condition(SingleColumnCondition("name", "John", ComparatorType.EQUAL))
cond.add_sub_condition(SingleColumnCondition("address", 'China', ComparatorType.EQUAL))
# Construct a request to read multiple rows.
request = BatchGetRowRequest()
# Add rows to be read from table_name. The 1 parameter specifies that the latest version
is read.
request.add(TableInBatchGetRowItem(table_name, rows_to_get, columns_to_get, cond, 1))
# Add rows to be read from notExistTable.
request.add(TableInBatchGetRowItem('notExistTable', rows_to_get, columns_to_get, cond, 1)
)
try:
       result = client.batch_get_row(request)
    print ('Result status: %s'%(result.is_all_succeed()))
    table_result_0 = result.get_result_by_table(table_name)
    table_result_1 = result.get_result_by_table('notExistTable')
    print ('Check first table\'s result:')
    for item in table_result_0:
        if item.is_ok:
            print ('Read succeed, PrimaryKey: %s, Attributes: %s' % (item.row.primary_key
, item.row.attribute_columns))
        else:
            print ('Read failed, error code: %s, error message: %s' % (item.error_code, i
tem.error_message))
    print ('Check second table\'s result:')
    for item in table_result_1:
        if item.is_ok:
            print ('Read succeed, PrimaryKey: %s, Attributes: %s' % (item.row.primary_key
, item.row.attribute_columns))
        else:
            print ('Read failed, error code: %s, error message: %s' % (item.error_code, i
tem.error_message))
# Client exceptions are caused by parameter errors or network exceptions.
except OTSClientError as e:
    print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.ge
t_error_message())
# Server exceptions are caused by parameter or throttling errors.
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())
```

For the detailed sample code, visit BatchGetRow@GitHub.

GetRange

You can call this operation to read rows of data in a forward or backward direction based on a specified primary key range.

You can also set the number of rows you want 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 the next primary key information are returned. You can also initiate a request to start from where the last operation left off and read the remaining rows based on the next primary key information returned by the previous operation.

The GetRange operation may stop and return data in the following situations:

- The amount of data scanned reaches 4 MB.
- The number of rows read reaches 5,000.
- The number of rows returned reaches the limit.
- All reserved read throughput is consumed. You have no reserved read throughput to read the next row of data.

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

- Operations

```
"""
Description: This operation reads data from multiple rows within the specified range.
table_name specifies the name of the table.
direction: the order for this query. Data type: string. Valid values: FORWARD and BACKWAR
D.
inclusive_start_primary_key: the primary key from which the read starts. If the specified
row exists, the row is included in the response.
exclusive_end_primary_key: the primary key column at which the read ends. The end primary
key column is excluded from the response even if the row exists.
columns_to_get: optional. This parameter specifies the names of columns you want to read.
If you do not specify this parameter, all columns are read. Type: list.
limit: optional. This parameter specifies the maximum number of rows you want to read. If
you do not specify this parameter, all rows are read.
column_filter: optional. This parameter specifies the filter conditions for columns. Only
rows that meet the condition are returned.
max_version is optional. This parameter specifies the maximum number of versions that can
be read. You must specify at least one of max_version and time_range.
time_range: optional. This parameter specifies the range of versions from which data is r
ead. You must specify at least one of max_version and time_range.
start_column: optional. This parameter specifies the column from which the read starts, w
hich is used to read wide rows.
end_column: optional. This parameter specifies the column at which the read ends, which i
s used to read wide rows.
token: optional. This parameter specifies the start column of a wide row you want to read
next time. The value of this parameter is returned by the last read request, and encoded
as binary data.
Response: the results that meet the conditions.
consumed: the number of CUs consumed by this operation. This parameter specifies an insta
nce of the tablestore.metadata.CapacityUnit class.
next_start_primary_key indicates the primary key from which you want to start the next ge
t_range operation. Type: dict.
row_list indicates the list of returned rows of data. Format: [Row, ...].
"""
def get_range(self, table_name, direction,
              inclusive_start_primary_key,
              exclusive_end_primary_key,
              columns_to_get=None,
              limit=None,
              column_filter=None,
              max_version=None,
              time_range=None,
              start_column=None,
              end_column=None,
              token = None):
```

- Parameters

Parameter	Description
table_name	The name of the table.

Parameter	Description
direction	The direction for reading. ◦ If the direction is set to FORWARD, the start primary key column value must be smaller than the end primary key column value. The rows are returned based on the primary key column values in ascending order. ◦ If the direction is set to BACKWARD, the start primary key column value must be greater than the end primary key column value. The rows are returned based on the primary key column values in descending order. Example: A table has two primary keys A and B where the value of A is smaller than that of B. If you set direction to forward, the rows whose primary key column values are greater than or equal to the value of A and smaller than the value of B are returned in ascending order. If you set direction to backward, the rows whose primary key column values are greater than the value of A and smaller than or equal to the value of B are returned in descending order.
inclusive_start_primary_key exclusive_end_primary_key	The start and end primary keys of the read operation. The start and end primary keys must be valid primary keys or virtual points of the INF_MIN or INF_MAX type. The number of columns for the virtual point must be the same as that of the primary key. INF_MIN indicates an infinitely small value of which all other values of other types are greater. INF_MAX indicates an infinitely large value of which all other values are smaller. ◦ inclusive_start_primary_key specifies the start primary key. If the row exists, the response contains this row. ◦ exclusive_end_primary_key specifies the end primary key. The response excludes the row no matter whether the row exists. Rows in tables are sorted based on the primary key column values in ascending order. The range for reading is a left-closed and right-open interval. When data is read in the forward direction, all rows whose primary key column values are greater than or equal to the start primary key column value and smaller than the end primary key column value are returned.
limit	The maximum number of returned rows. This parameter value must be greater than 0. An operation stops when the maximum number of rows is returned in the forward or backward direction, even if part of rows within the specified range are not returned. At this time, you can record the next_start_primary_key value in the response for the next reading.

Parameter	Description
columns_to_get	The collection of columns you want to read. The columns can be primary key columns or attribute columns. If this parameter is not set, rows that contain all columns are returned. **Note** ◦ If you query a row of data, the system returns the data in all columns of the row. You can set the columns_to_get parameter to return the data in specified columns. If you add col0 and col1 to columns_to_get, only the col0 and col1 values are returned. ◦ If the primary key set for a row is within the specified range, but the row does not contain the specified columns, the response does not contain the row of data. ◦ If columns_to_get and column_filter are used at the same time, the system first obtains the columns specified by columns_to_get and then filters the returned columns.
max_version	The maximum number of versions you want to read. **Note** You must specify at least one of max_version and time_range. ◦ If you specify only max_version, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, all data within a range or a version of data is returned. ◦ If you specify both max_version and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
time_range	Reads data within a range of versions or a version of data. For more information, see TimeRange. **Note** You must specify at least one of max_version and time_range. ◦ If you specify only max_version, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, all data within a range or a version of data is returned. ◦ If you specify both max_version and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ To query data within a range, you must set start_time and end_time. start_time specifies the start timestamp. end_time specifies the end timestamp. The time range is a left-closed and right-open interval, which is [start_time, end_time). ◦ To query data within a range of versions, set specific_time. specific_time specifies a specific timestamp. Only one of specific_time and [start_time, end_time) is required. Valid values: [0, INT64.MAX). Unit: milliseconds.
column_filter	Filters read results on the server to return rows that meet the conditions in the filter. For more information, see Filter. **Note** If columns_to_get and column_filter are used at the same time, the system first obtains the columns specified by columns_to_get and then filters the returned columns.
next_start_primary_key	The start primary key for the next reading. The value of next_start_primary_key can be used to determine whether all data is read. ◦ If the value of next_start_primary_key is not empty in the response, the next_start_primary_key value can be used as the start primary key for the next GetRange operation. ◦ If the value of next_start_primary_key is empty in the response, all data within the range is returned.

- Examples

  The following code provides an example on how to read data within a specified range:

```
# Specify the start primary key.
inclusive_start_primary_key = [('uid', INF_MIN), ('gid', INF_MIN)]
# Specify the end primary key.
exclusive_end_primary_key = [('uid', INF_MAX), ('gid', INF_MAX)]
# Query all columns.
```

```
columns_to_get = []
# Set limit to 90 to return a maximum of 90 rows of data. If a total of 100 rows are to b
e returned, you can set limit to 90 in the first query. The next_start_primary_key value
is not None.
limit = 90
# Configure the filter conditions.
cond = CompositeColumnCondition(LogicalOperator.AND)
cond.add_sub_condition(SingleColumnCondition("address", 'China', ComparatorType.EQUAL))
cond.add_sub_condition(SingleColumnCondition("age", 50, ComparatorType.LESS_THAN))
try:
    # Call the get_range operation.
    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,
        time_range = (1557125059000, 1557129059000) # The start_time value is equal to or
greater than 1557125059000. The end_time value is smaller than 1557129059000.
    )
    all_rows = []
    all_rows.extend(row_list)
    # If the next_start_primary_key value is not None, the read continues until all remai
ning data is read.
    while next_start_primary_key is not None:
        inclusive_start_primary_key = next_start_primary_key
        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)
    # Display the primary key and attribute columns.
    for row in all_rows:
        print(row.primary_key, row.attribute_columns)
    print('Total rows: ', len(all_rows))
# Client exceptions are caused by parameter errors or network exceptions.
except OTSClientError as e:
    print
    "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.get_erro
r_message())
# Server exceptions are caused by parameter or throttling errors.
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())
```

For the detailed sample code, visit Get Range@GitHub.

4.7. Search Index

4.7.1. Create search indexes

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A data table whose time_to_live is set to -1 and max_versions is set to 1 is created.

Parameters

When you create a search index, you must specify table_name, index_name, and schema. You must specify field_schemas, index_setting, and index_sort in schema. The following table describes the parameters.

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.

Parameter	Description
field_schemas	The list of field schemas. You can configure the following parameters for each field schema: • field_name: required. This parameter specifies the name of the field in the search index. The value is a column name. Type: String. A field in a search index can be a primary key column or an attribute column. • field_type: required. This parameter specifies the type of a field. Use FieldType.XXX to set the type. For more information, see Data types of column values. • is_array: optional. This parameter specifies whether the value is an array. Type: Boolean. If you set this parameter to true, the column stores data as an array. Data written to the column must be a JSON array. Example: ["a","b","c"]. Nested values are an array. If you set field_type to Nested, skip this parameter. • index: optional. This parameter specifies whether to create an index for the column. Type: Boolean. By default, this parameter is set to true, and Tablestore creates an inverted index or spatial index for the column. If this parameter is set to false, Tablestore does not create indexes for the column. • analyzer: optional. This parameter specifies the type of analyzer to use. If field_type is set to Text, you can set this parameter. If you do not specify this parameter, single-word tokenization is used. For more information about tokenization, see Tokenization. • enable_sort_and_agg: optional. This parameter specifies whether to enable the sorting and aggregation features. Type: Boolean. Before you enable the sorting feature, you need to set enable_sort_and_agg to true for a field. For more information about sorting, see Sorting and pagination. Notice • store: optional. This parameter specifies whether to store the value of the field in the search index. Type: Boolean. If you set the value to true, you can read the value of the field directly from the search index without querying the entire table. This improves query performance. • sub_field_schemas: optional. This parameter specifies the list of field schemas for subfields. If the column is a Nested column, you must specify this parameter to configure the index types of subcolumns in the Nested column.

Parameter	Description
index_setting	The settings of the search index, including routing_fields. routing_fields: optional. This parameter specifies custom routing fields. You can specify some primary key columns as routing fields. Tablestore 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	The presorting settings of the search index, including sorters. If indexSort is left empty, data is sorted by primary key. **Note** You can skip the presorting settings for search indexes that contain the Nested field type. sorters: required. This parameter specifies the presorting method for the search index. PrimaryKeySort and FieldSort are supported. For more information about sorting, see Sorting and pagination. • PrimaryKeySort: Data is sorted by primary key. You can configure the following parameter for PrimaryKeySort: sort_order: the sort order. Data can be sorted in ascending or descending order. Default value: SortOrder.ASC. • FieldSort: Data is sorted by field value. You can configure the following parameters for FieldSort: You can presort field values only when a search index is created and the sorting and aggregation features are enabled for fields in the search index. ◦ field_name: the name of the field to sort. ◦ sort_order: the sort order. Data can be sorted in ascending or descending order. Default value: SortOrder.ASC. ◦ sort_mode: the sorting method used when the field has multiple values.

Examples

```
field_a = FieldSchema('k', FieldType.KEYWORD, index=True, enable_sort_and_agg=True, store=T
rue)
field_b = FieldSchema('t', FieldType.TEXT, index=True, store=True, analyzer=AnalyzerType.SI
NGLEWORD)
field_c = FieldSchema('g', FieldType.GEOPOINT, 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 # can not set index sort if there is any nested field.
#index_sort = Sort(sorters=[PrimaryKeySort(SortOrder.ASC)])
index_meta = SearchIndexMeta(fields, index_setting=index_setting, index_sort=index_sort)
client.create_search_index(table_name, index_name, index_meta)
```

4.7.2. 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.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.

Examples

```
index_meta, sync_stat = client.describe_search_index(table_name, index_name)
print json.dumps(index_meta, default=lambda x:x.__dict__, indent=2) // Display the index_me
ta information of the search index.
print json.dumps(sync_stat, default=lambda x:x.__dict__, indent=2) // Display the synchroni
zation status of data in the search index.
```

4.7.3. List search indexes

You can call the ListSearchIndex operation to obtain indexes created in the current instance or associated with a table.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

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

4.7.4. Delete search indexes

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.

Examples

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

4.7.5. Term query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
query_type	The query type. To use term query, set this parameter to TermQuery.
field_name	The name of the field used to match the query conditions.
term	The keyword used to match the field values when you perform a term query. This word is not tokenized. Instead, the whole word is used to match the field values. If the type of a field is TEXT, Tablestore tokenizes the string and exactly matches tokens. For example, if the value of a TEXT field in a row is "tablestore is cool", the value can be tokenized into "tablestore", "is", and "cool". The row can meet query conditions when you set "tablestore", "is", or "cool" to the keyword to match the values of the TEXT field.
table_name	The name of the base table.
index_name	The name of the search index.
limit	The maximum number of rows that the current query returns. To query just the number of matched rows without any detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to False, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
columns_to_get	Specifies whether to return all columns of each matched row. You can configure return_type and column_names for this parameter. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Use term query to query the rows whose k column values exactly match 'key000'.

- Perform term query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a term query, SearchResponse objects are returned. The following code provides a request sample:

```
query = TermQuery('k', 'key000')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  You can use the following sample request to return results of the Tuple type:

```
query = TermQuery('k', 'key000')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform term query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a term query, results of the Tuple type are returned. The following code provides a request sample:

```
query = TermQuery('k', 'key000')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.6. Terms query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
query_type	The query type. To use terms query, set this parameter to TermsQuery.
field_name	The name of the field used to match the query conditions.
terms	The keywords used to match the field values when you perform a terms query. A row meets the query conditions when the value of the specified field exactly matches at least one keyword.
table_name	The name of the base table.
index_name	The name of the search index.
limit	The maximum number of rows that the current query returns. To query just the number of matched rows without any detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to False, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
columns_To_get	Specifies whether to return all columns of each matched row. You can configure return_type and column_names for this parameter. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Use terms query to query the rows whose k column values match one of the following keywords: 'key000', 'key100', 'key888', 'key999', 'key908', and 'key1000'.

- Perform terms query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a terms query, SearchResponse objects are returned. The following code provides a request sample:

```
query = TermsQuery('k', ['key000', 'key100', 'key888', 'key999', 'key908', 'key1000'])
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

You can use the following sample request to return results of the Tuple type:

```
query = TermsQuery('k', ['key000', 'key100', 'key888', 'key999', 'key908', 'key1000'])
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform terms query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a terms query, results of the Tuple type are returned. The following code provides a request sample:

```
query = TermsQuery('k', ['key000', 'key100', 'key888', 'key999', 'key908', 'key1000'])
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.7. Match all query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
query	The query type. To use match all query, set this parameter to MatchAllQuery.
table_name	The name of the table.
index_name	The name of the search index.
limit	The maximum number of rows that the current query returns. To query the number of matched rows without detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.

Parameter	Description
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
columns_to_get	Specifies whether to return all columns of each matched row. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Use match all query to query the total number of rows in the table.

- Perform match all query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a match all query, SearchResponse objects are returned. The following code provides a request sample:

```
query = MatchAllQuery()
all_rows = []
next_token = None
while not all_rows or next_token:
    search_response = client.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED))
    all_rows.extend(search_response.rows)
for row in all_rows:
    print row
print 'Total rows:', len(all_rows)
```

  You can use the following sample request to return results of the Tuple type:

```
query = MatchAllQuery()
all_rows = []
next_token = None
while not all_rows or next_token:
    rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client
.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED)).v1_response()
    all_rows.extend(rows)
for row in all_rows:
    print row
print 'Total rows:', len(all_rows)
```

- Perform match all query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a match all query, results of the Tuple type are returned. The following code provides a request sample:

```
query = MatchAllQuery()
all_rows = []
next_token = None
while not all_rows or next_token:
    rows, next_token, total_count, is_all_succeed = client.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED))
    all_rows.extend(rows)
for row in all_rows:
    print row
print 'Total rows:', len(all_rows)
```

4.7.8. Match query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description

Parameter	Description
field_name	The name of the column you want to query. Match query applies to TEXT columns.
text	The keyword used to match the column values when you perform a match query. If the column to query is a TEXT column, the keyword is tokenized into multiple tokens based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you set the analyzer type to single-word tokenization and use "this is" as a search string, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
query	The query type. To use match query, set this parameter to MatchQuery.
table_name	The name of the table.
index_name	The name of the search index.
limit	The maximum number of rows that the current query returns. To query the number of matched rows without detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
operator	The logical operator. By default, OR is used as the logical operator, which indicates that a row matches the query conditions when one of the tokens is matched. If you set the operator to AND, the row data meets the query condition only when all tokens are in the column values.
minimum_should_match	The minimum number of matched tokens contained in a column value. A row is returned only when the value of the queried column in the row contains at least the minimum number of matched tokenized keywords. **Note** minimum_should_match must be used together with the OR logical operator.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to False, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.

Parameter	Description
columns_to_get	Specifies whether to return all columns of each matched row. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Use match query to query the rows in which the values of the t columns approximately match 'this is 0'.

- Perform match query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a match query, SearchResponse objects are returned. The following code provides a request sample:

```
query = MatchQuery('t', 'this is 0')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  You can use the following sample request to return results of the Tuple type:

```
query = MatchQuery('t', 'this is 0')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform match query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a match query, results of the Tuple type are returned. The following code provides a request sample:

```
query = MatchQuery('t', 'this is 0')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.9. Match phrase query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
field_name	The name of the column you want to query. Match phrase query applies to TEXT columns.
text	The keyword used to match the column values when you perform a phase query. If the column to query is a TEXT column, the keyword is tokenized into multiple tokens based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you query the phrase "this is", results such as "..., this is tablestore" and "this is a table" are returned. Query results such as "this table is ..."or "is this a table" are not returned.
query	The query type. To use match phrase query, set this parameter to MatchPhraseQuery.
table_name	The name of the base table.
index_name	The name of the search index.
limit	The maximum number of rows that the current query returns. To query just the number of matched rows without any detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to False, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.

Parameter	Description
columns_to_get	Specifies whether to return all columns of each matched row. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Use match phrase query to query the rows whose t column values match the 'this is' phrase in sequence.

- Perform match phrase query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a match phrase query, SearchResponse objects are returned. The following code provides a request sample:

```
query = MatchPhraseQuery('t', 'this is')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  You can use the following sample request to return results of the Tuple type:

```
query = MatchPhraseQuery('t', 'this is')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform match phrase query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a match phrase query, results of the Tuple type are returned. The following code provides a request sample:

```
query = MatchPhraseQuery('t', 'this is')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.10. Prefix query

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

Prerequisites

.

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
field_name	The name of the column used to match the query conditions.
prefix	The prefix. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that contains the specified prefix.
query	The type of the query. Set the query type to PrefixQuery.
table_name	The name of the table.
index_name	The name of the search index.
limit	The maximum number of rows that the current query returns. To query the number of matched rows without detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to False, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
columns_to_get	Specifies whether to return all columns of each matched row. You can configure return_type and column_names for this parameter. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Use prefix query to query the rows in which the values of the k columns are prefixed with 'key00'.

- Perform prefix query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a prefix query, SearchResponse objects are returned. The following code provides a request sample:

```
query = PrefixQuery('k', 'key00')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

You can use the following sample request to return results of the Tuple type:

```
query = PrefixQuery('k', 'key00')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform prefix query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a prefix query, results of the Tuple type are returned. The following code provides a request sample:

```
query = PrefixQuery('k', 'key00')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.11. Range query

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
field_name	The name of the field used to match the query conditions.
range_from	The value from which the query starts.
range_to	The value with which the query ends.
include_lower	Specifies whether to include the value of range_from in the response. The parameter is of the Boolean type.
include_upper	Specifies whether to include the value of range_to in the response. The parameter is of the Boolean type.

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
query	The query type. To use range query, set this parameter to RangeQuery.
limit	The maximum number of rows that the current query returns. To query just the number of matched rows without any detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to False, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
columns_to_get	Specifies whether to return all columns of each matched row. You can configure return_type and column_names for this parameter. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Use range query to query the rows whose k column values range from 'key100' to 'key200'.

- Perform range query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a range query, SearchResponse objects are returned. The following code provides a request sample:

```
query = RangeQuery('k', 'key100', 'key200', include_lower=False, include_upper=False)
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  You can use the following sample request to return results of the Tuple type:

```
query = RangeQuery('k', 'key100', 'key200', include_lower=False, include_upper=False)
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform range query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a range query, results of the Tuple type are returned. The following code provides a request sample:

```
query = RangeQuery('k', 'key100', 'key200', include_lower=False, include_upper=False)
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.12. Wildcard query

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

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
query	The type of the query. Set the query type to WildcardQuery.
field_name	The name of the column.
value	The string that contains wildcards. The string cannot exceed 32 characters in length.
table_name	The name of the data table.
index_name	The name of the search index.

Parameter	Description
limit	The maximum number of rows that you want the current query to return.
get_total_count	Specifies whether to return the total number of matched rows. Default value: False. If you set the get_total_count parameter to true, the query performance is compromised.
ColumnsToGet	Specifies whether to return all columns of each matched row. • If you set the return_type parameter to ColumnReturnType.SPECIFIED, you must specify the columns that you want to return. • If you set the return_type parameter to ColumnReturnType.ALL, all columns are returned. • If you set the return_type parameter to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

Search the table for rows in which the value of the k column matches 'key00*'.

- Perform a wildcard query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a wildcard query, a SearchResponse object is returned. The following code provides a sample request:

  ```
  query = WildcardQuery('k', 'key00*')
  search_response = client.search(
      table_name, index_name,
      SearchQuery(query, limit=100, get_total_count=True),
      ColumnsToGet(return_type=ColumnReturnType.ALL)
  )
  ```

  You can use the following sample request to return results of the Tuple type:

  ```
  query = WildcardQuery('k', 'key00*')
  rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
  rch(
      table_name, index_name,
      SearchQuery(query, limit=100, get_total_count=True),
      ColumnsToGet(return_type=ColumnReturnType.ALL)
  ).v1_response()
  ```

- Perform a wildcard query by using a version of Tablestore SDK for Python that is earlier than V5.2.1

  By default, if you use a version of Tablestore SDK for Python that is earlier than V5.2.1 to perform a wildcard query, results of the Tuple type are returned. The following code provides a sample request:

```
query = WildcardQuery('k', 'key00*')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.13. Exists query

Exists query is also called NULL query or NULL-value query. This query is used in sparse data to determine whether a column of a row exists. For example, you can query the rows in which the value of the address column is not empty.

> **Note**
>
> - If you want to check whether a column contains empty values, you must use ExistsQuery together with must_not_queries of BoolQuery.
> - If one of the following conditions is met, the system considers that a column does not to exist. In this example, the city column is used.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the city column does not exist in the data table, the search index considers that the city column does not exist.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the value of the city column is an empty array in the data table ("city" = "[]"), the search index considers that the city column does not exist.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
field_name	The name of the column.
query	The query type. Set the value to ExistsQuery.
get_total_count	Specifies whether to return the total number of rows that meet the query conditions. The default value of this parameter is false, which specifies that the total number of rows that meet the query conditions is not returned. If you set this parameter to true, the query performance is compromised.
table_name	The name of the data table.

Parameter	Description
index_name	The name of the search index.
columns_to_get	Specifies whether to return all columns of each row that meets the query conditions. You can configure return_type and column_names for this parameter. • If you set return_type to ColumnReturnType.SPECIFIED, you can use column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

The following sample code shows how to query the rows in which the city column is not empty:

- Perform an exists query by using Tablestore SDK for Python V5.2.1 or later

  If you use Tablestore SDK for Python V5.2.1 or later to perform an exists query, a SearchResponse object is returned by default. The following code shows a sample request:

  ```
  query = ExistsQuery("city")
  search_response = client.search(
      table_name, index_name,
      SearchQuery(query, limit=100, get_total_count=True),
      ColumnsToGet(return_type=ColumnReturnType.ALL)
  )
  ```

  You can use the following sample request to obtain results of the Tuple type:

  ```
  query = ExistsQuery("city")
  rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
  rch(
      table_name, index_name,
      SearchQuery(query, limit=100, get_total_count=True),
      ColumnsToGet(return_type=ColumnReturnType.ALL)
  ).v1_response()
  ```

- Perform an exists query by using a version of Tablestore SDK for Python that is earlier than V5.2.1

  If you use a version of Tablestore SDK for Python that is earlier than V5.2.1 to perform an exists query, results of the Tuple type are returned by default. The following code shows a sample request:

  ```
  query = ExistsQuery("city")
  rows, next_token, total_count, is_all_succeed = client.search(
      table_name, index_name,
      SearchQuery(query, limit=100, get_total_count=True),
      ColumnsToGet(return_type=ColumnReturnType.ALL)
  )
  ```

4.7.14. Geo query

Geo query includes geo-distance query, geo-bounding box query, and geo-polygon query.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Geo-distance query

You can use geo-distance query to specify a circular geographical area consisting of a central point and radius as a filtering condition in a query. Tablestore returns the rows where the values of a specified column are within the circular geographical area.

- Parameters

Parameter	Description
field_name	The name of the column used to match the query conditions. The specified column must be a GEOPOINT column.
center_point	The location of the central point. The value of this parameter is of the GEOPOINT data type in the following format: `latitude,longitude`. Valid values of latitude: [-90,+90]. Valid values of longitude: [-180,+180]. Example: `35.8,-45.91`.
distance	The radius of the circular geographical area. The value of this parameter is of the DOUBLE data type. Unit: meters.
query	The query statement for the search index. The query type. To use geo-distance query, set this parameter to GeoDistanceQuery.
table_name	The name of the table.
index_name	The name of the search index.

- Examples

  The following code provides an example on how to query rows where the values of the g column are within the circular geographical area whose central point is '32.5,116.5' and radius is 300,000 meters:

```
query = GeoDistanceQuery('g', '32.5,116.5', 300000)
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

Geo-bounding box query

You can use geo-distance query to specify a rectangular geographical area. Tablestore returns the rows where the values of a specified column are within the rectangular geographical area.

- Parameters

Parameter	Description
field_name	The name of the column used to match the query conditions. The specified column must be a GEOPOINT column.
top_left	The location of the top-left corner of the rectangle box.
bottom_right	The location of the bottom-right corner of the rectangular box. A rectangular geographical area can be specified by a top-left corner and a bottom-right corner. The value of this parameter is in the following format: `latitude,longitude`. Valid values of latitude: [-90,+90]. Valid values of longitude: [-180,+180]. Example: `35.8,-45.91`.
query	The query statement for the search index. The query type. To use geo-bounding box query, set this parameter to GeoBoundingBoxQuery.
table_name	The name of the table.
index_name	The name of the search index.

- Examples

  The following code provides an example on how to query rows where the values of the g column are within the rectangular geographical area whose top-left corner is at '30.9,112.0' and bottom-right corner at '30.2,119.0'.

```
query = GeoBoundingBoxQuery('g', '30.9,112.0', '30.2,119.0')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

Geo-polygon query

You can use geo-polygon query to specify a polygonal geographical area. Tablestore returns the rows where the values of a specified column are within the polygonal geographical area.

- Parameters

Parameter	Description
field_name	The name of the column used to match the query conditions. The specified column must be a GEOPOINT column.

Parameter	Description
points	The coordinates of the points that compose the polygon. You can specify a polygon by using multiple coordinates in the following format: `latitude,longitude`.Valid values of latitude: [-90,+90]. Valid values of longitude: [-180,+180]. Example: `35.8,-45.91`.
query	The query statement for the search index. The query type. To use geo-polygon query, set this parameter to GeoPolygonQuery.
table_name	The name of the table.
index_name	The name of the search index.

- Examples

  The following code provides an example on how to query rows where the values of the g column are within the polygonal geographical area that consists of the coordinates of '30.9,112.0', '30.5,115.0', '30.3, 117.0', and '30.2,119.0':

```
query = GeoPolygonQuery('g', ['30.9,112.0', '30.5,115.0', '30.3, 117.0', '30.2,119.0'])
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.15. Boolean query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
must_queries	The list of subqueries that the query results must match. This parameter is equivalent to the AND operator.
must_not_queries	The list of subqueries that the query results must not match. This parameter is equivalent to the NOT operator.

Parameter	Description
filter_queries	The list of subqueries. Only rows that match all subfilters are returned. filter is similar to query except that filter does not calculate the relevance score based on the number of subfilters that the row matches.
should_queries	The list of subqueries that the query results can or cannot match. This parameter is equivalent to the OR operator. Only rows that meet the minimum number of subquery conditions specified by should_queries are returned. A higher overall relevance score indicates that more subquery conditions specified by should_queries are met.
minimum_should_match	The minimum number of subquery conditions specified by should_queries that the rows must meet. If no other subquery conditions except the subquery conditions that are specified by should_queries are specified, the default value of the minimum_should_match parameter is 1. If other subquery conditions, such as subquery conditions specified by must_queries, must_not_queries, and filter_queries, are specified, the default value of the minimum_should_match parameter is 0.
table_name	The name of the data table.
index_name	The name of the search index.

Examples

- Perform a Boolean query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a Boolean query, SearchResponse objects are returned. The following code provides a sample request:

```
# k > 'key100' and (l > 110 and l < 200) and not (k = 'key121')
# and should_queries(k > 'key120' or l < 300, minimum_should_match=2)
bool_query = BoolQuery(
    must_queries=[
        RangeQuery('k', range_from='key100', include_lower=False),
        // Set the query type to BoolQuery.
        BoolQuery(
            // Set subqueries that the rows must match.
            must_queries=[
                RangeQuery('l', range_from=110, include_lower=False),
                RangeQuery('l', range_to=200, include_upper=False)
            ],
        )
    ],
    // Set subqueries that the rows must not match.
    must_not_queries=[
        TermQuery('k', 'key121')
    ],
    should_queries=[
        RangeQuery('k', range_from='key120', include_lower=False),
        RangeQuery('l', range_to=300, include_upper=130)
    ],
    minimum_should_match=2
)
// Construct a Boolean query by specifying query parameters, including sort, limit, and g
et_total_count.
search_response = client.search(
    table_name, index_name,
    SearchQuery(
        bool_query,
        sort=Sort(sorters=[FieldSort('l', SortOrder.ASC)]),
        limit=100,
        get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

You can use the following sample request to return results of the Tuple type:

```
# k > 'key100' and (l > 110 and l < 200) and not (k = 'key121')
# and should_queries(k > 'key120' or l < 300, minimum_should_match=2)
bool_query = BoolQuery(
    must_queries=[
        RangeQuery('k', range_from='key100', include_lower=False),
        // Set the query type to BoolQuery.
        BoolQuery(
            // Set subqueries that the rows must match.
            must_queries=[
                RangeQuery('l', range_from=110, include_lower=False),
                RangeQuery('l', range_to=200, include_upper=False)
            ],
        )
    ],
    // Set subqueries that the rows must not match.
    must_not_queries=[
        TermQuery('k', 'key121')
    ],
    should_queries=[
        RangeQuery('k', range_from='key120', include_lower=False),
        RangeQuery('l', range_to=300, include_upper=130)
    ],
    minimum_should_match=2
)
// Construct a Boolean query by specifying query parameters, including sort, limit, and g
et_total_count.
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(
        bool_query,
        sort=Sort(sorters=[FieldSort('l', SortOrder.ASC)]),
        limit=100,
        get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform a Boolean query by using Tablestore SDK for Python of a version earlier than V5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than V5.2.1 to perform a Boolean query, results of the Tuple type are returned. The following code provides a sample request:

```
# k > 'key100' and (l > 110 and l < 200) and not (k = 'key121')
# and should_queries(k > 'key120' or l < 300, minimum_should_match=2)
bool_query = BoolQuery(
    must_queries=[
        RangeQuery('k', range_from='key100', include_lower=False),
        // Set the query type to BoolQuery.
        BoolQuery(
            // Set subqueries that the rows must match.
            must_queries=[
                RangeQuery('l', range_from=110, include_lower=False),
                RangeQuery('l', range_to=200, include_upper=False)
            ],
        )
    ],
    // Set subqueries that the rows must not match.
    must_not_queries=[
        TermQuery('k', 'key121')
    ],
    should_queries=[
        RangeQuery('k', range_from='key120', include_lower=False),
        RangeQuery('l', range_to=300, include_upper=130)
    ],
    minimum_should_match=2
)
// Construct a Boolean query by specifying query parameters, including sort, limit, and g
et_total_count.
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(
        bool_query,
        sort=Sort(sorters=[FieldSort('l', SortOrder.ASC)]),
        limit=100,
        get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.16. Nested query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
path	The path of the nested column. The path is similar to the tree structure. For example, news.title indicates the title subcolumn in the news column of the nested type.
query	The query on the subcolumn in the nested column. The query can be of any query type.
score_mode	Specifies which value is used to calculate the score when a column contains multiple values.
table_name	The name of the table.
index_name	The name of the search index.

Examples

Use nested query to query the value of the n.n1 column that is greater than or equal to 100 and smaller than or equal to 300.

- Perform nested query by using Tablestore SDK for Python V5.2.1 or later

  By default, if you use Tablestore SDK for Python V5.2.1 or later to perform a nested query, SearchResponse objects are returned. The following code provides a request sample:

```
nested_query = RangeQuery('n.n1', range_from=100, range_to=300, include_lower=True, inclu
de_upper=True)
query = NestedQuery('n', nested_query)
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  You can use the following sample request to return results of the Tuple type:

```
nested_query = RangeQuery('n.n1', range_from=100, range_to=300, include_lower=True, inclu
de_upper=True)
query = NestedQuery('n', nested_query)
rows, next_token, total_count, is_all_succeed, agg_result, group_by_results = client.sear
ch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- Perform nested query by using Tablestore SDK for Python of a version earlier than 5.2.1

  By default, if you use Tablestore SDK for Python of a version earlier than 5.2.1 to perform a nested query, results of the Tuple type are returned. The following code provides a request sample:

```
nested_query = RangeQuery('n.nl', range_from=100, range_to=300, include_lower=True, inclu
de_upper=True)
query = NestedQuery('n', nested_query)
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

4.7.17. Sorting and pagination

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

Index presorting

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

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

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

Specify a sorting method

Sorting can be enabled only for fields for which enable_sort_and_agg is set to True.

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

- ScoreSort

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

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

  ```
  sort = Sort(
      sorters=[ScoreSort(sort_order=SortOrder.DESC)]
  )
  client.search(
      table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
  ), ColumnsToGet(return_type=ColumnReturnType.ALL)
  )
  ```

- PrimaryKeySort

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

```
sort = Sort(
    sorters=[PrimaryKeySort(sort_order=SortOrder.DESC)]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
) = PrimaryKeySort(sort_order=SortOrder.DESC)
```

- FieldSort

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

  Sort the query result based on a non-nested field:

```
sort = Sort(
    sorters=[FieldSort('l', SortOrder.ASC)]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  Sort the query result based on a nested field:

```
sort = Sort(
    sorters=[
        FieldSort(
            'n.nl',
            sort_order=SortOrder.ASC,
            nested_filter=NestedFilter('n', RangeQuery('n.nl', range_from=150, range_to=2
00))
        )
    ]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

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

```
sort = Sort(
    sorters=[
        FieldSort('a', SortOrder.ASC),
        FieldSort('b', SortOrder.ASC)
    ]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

- GeoDistanceSort

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

```
sort = Sort(
    sorters=[GeoDistanceSort('g', ['32.5,116.5', '32.0,116.0'], sort_order=SortOrder.DESC
, sort_mode=SortMode.MAX)]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

Specify a pagination method

You can use limit and offset or use tokens to split returned query results into pages.

- Use the limit and offset parameters

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

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

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

  ```
  query = RangeQuery('k', 'key100', 'key500', include_lower=False, include_upper=False)
  search_response = client.search(
      table_name, index_name,
      SearchQuery(query, offset=100, limit=100, get_total_count=True),
      ColumnsToGet(return_type=ColumnReturnType.ALL)
  )
  ```

- Use a token

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

  If Tablestore is not able to complete reading data that meets the filter condition, the server returns next_token. You can use next_token to continue reading the subsequent data.

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

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

> **Notice** Search indexes of the nested type do not support IndexSort. If you use a search index of the nested type to query data and require pagination, you must specify the sorting method to return data in the query conditions. Otherwise, Tablestore does not return next_token when all data that meets the query conditions is not completely read.

```
query = MatchAllQuery()
all_rows = []
next_token = None
# first round
search_response = client.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED))
all_rows.extend(search_response.rows)
# loop
while search_response.next_token:
    search_response = client.search(table_name, index_name,
        SearchQuery(query, next_token=search_response.next_token, sort=None, limit=100, g
et_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED))
    all_rows.extend(search_response.rows)
print('Total rows:%d' % len(all_rows))
```

4.7.18. Aggregation

You can use aggregation to obtain the minimum value, maximum value, sum, and average of columns and the count and distinct count of rows. You can also use aggregation to group the obtained results by field value, range, geographical location, or filter. You can also use multiple aggregations to perform complex queries.

> **Note** Tablestore SDK for Python V5.2.1 and later support the aggregation feature.

Minimum value

The aggregation method that can be used to return the minimum value of a field. This method can be used in a similar manner as the SQL MIN function.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on its name.
field	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.

Parameter	Description
missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ○ If the missing parameter is not specified, the row is ignored. ○ If the missing parameter is set, the value of this parameter is used as the field value of the row.

- Examples

  The following code provides an example on how to use aggregation to obtain the lowest score from the scores of people aged 18:

```
query = TermQuery('age', 18)
agg = Min('score', name = 'min')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=0, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

Maximum value

The aggregation method that can be used to return the maximum value of a field. This method can be used in a similar manner as the SQL MAX function.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on its name.
field	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ○ If the missing parameter is not specified, the row is ignored. ○ If the missing parameter is set, the value of this parameter is used as the field value of the row.

- Examples

  The following code provides an example on how to use aggregation to obtain the highest score from the scores of people aged 18. By default, if a person has no score, a value of 0 is used as the score for aggregation.

```
query = TermQuery('age', 18)
agg = Max('score', missing_value = 0, name = 'max')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=0, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

Sum

The aggregation method that can be used to return the sum of all values for a numeric field. This method can be used in a similar manner as the SQL SUM function.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on its name.
field	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ◦ If the missing parameter is not specified, the row is ignored. ◦ If the missing parameter is set, the value of this parameter is used as the field value of the row.

- Examples

  The following code provides an example on how to use aggregation to obtain a sum of the scores of people aged 18:

```
query = TermQuery('age', 18)
agg = Sum('score', name = 'sum')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

Average

The aggregation method that can be used to return the average of all values for a numeric field. This method is used in a similar manner as the SQL AVG function.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on its name.
field	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ○ If the missing parameter is not specified, the row is ignored. ○ If the missing parameter is set, the value of this parameter is used as the field value of the row.

- Examples

  The following code provides an example on how to use aggregation to obtain an average of the scores of people aged 18:

```
query = TermQuery('age', 18)
agg = Avg('score', name = 'avg')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

Count

The aggregation method that can be used to return the total number of values for a specified field or the total number of rows in a table. This method can be used in a similar manner as the SQL COUNT function.

> **Note** You can use one of the following methods to query the total number of rows in a table or the total number of rows that match the query conditions:
>
> - Use the count feature of aggregation. Set the count parameter to * in the request.
> - Use the query feature to obtain the number of rows that match the query conditions. Set the setGetTotalCount parameter to true in the query. Use MatchAllQuery to obtain the total number of rows in a table.
>
> You can use the name of a column as the value of the count expression to query the number of rows that contain the column in a table. This method is suitable for scenarios that involve sparse columns.

- Parameters

Parameter	Description

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on the name.
field	The name of the field used to perform the aggregation operation. Only the LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT data types are supported.

- Examples

  The following code provides an example on how to use aggregation to obtain the number of people aged 18 that have taken the exam and have scores.

```
query = TermQuery('age', 18)
agg = Count('score', name = 'count')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

Distinct count

The aggregation method that can be used to return the number of distinct values for a field. This method can be used in a similar manner as the SQL COUNT(DISTINCT) function.

> **Note** The number of distinct values is an approximate number.
> - If the total number of rows before the distinct count feature is used is less than 10,000, the calculated result is an exact value.
> - If the total number of rows before the distinct count feature is used is greater than or equal to 100 million, the error rate is approximately 2%.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on its name.
field	The name of the field used to perform the aggregation operation. Only the LONG, DOUBLE, BOOLEAN, KEYWORD, and GEOPOINT data types are supported.
missing	The default value for the field used for the aggregation operation on a row when the field value is empty. ○ If the Missing parameter is not specified, the row is ignored. ○ If the Missing parameter is set, the value of this parameter is used as the field value of the row.

- Examples

  The following code provides an example on how to use aggregation to obtain the number of distinct names used by people aged 18:

```
query = TermQuery('age', 18)
agg = DistinctCount('name', name = 'distinct_name')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

Group by field value

The aggregation method that can be used to group query results based on field values. The values that are the same are grouped together. The identical value of each group and the number of identical values in each group are returned.

> **Note** The calculated number may be different from the actual number if the number of values in a group is very large.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on the name.
field	The name of the field used to perform the aggregation operation. Only the LONG, DOUBLE, BOOLEAN, and KEYWORD data types are supported.
size	The number of returned groups.
group_by_sort	The sorting rules for items in a group. By default, group items are sorted in descending order. When you set multiple sorting rules, data is sorted based on the order in which the rules are added. Supported parameters: ◦ Sort by value in ascending alphabetical order ◦ Sort by value in descending alphabetical order ◦ Sort by row count in ascending order ◦ Sort by row count in descending order ◦ Sort based on values obtained from the sub-aggregation in ascending order ◦ Sort based on values obtained from the sub-aggregation in descending order

Parameter	Description
sub_aggs and sub_group_bys	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results. ○ Scenarios Query the number of products in each category and the maximum and minimum product prices in each category. ○ Method First group query results by product category, and then add two sub-aggregations to obtain the maximum and minimum product prices in each category. ○ Result ■ Fruits: The number is 5. The maximum price is 15. The minimum price is 3. ■ Toiletries: The number is 10. The maximum price is 98. The minimum price is 1. ■ Electronic devices: The number is 3. The maximum price is 8,699. The minimum price is 2,300. ■ Other products: The number is 15. The maximum price is 1,000. The minimum price is 80.

- Example 1

  The following code provides an example on how to group people aged 18 by score and obtain the top 10 common scores and the number of people in each group.

```
query = TermQuery('age', 18)
group_by = GroupByField('score', size = 10)
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=20, group_bys
= [group_by]),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("key:%s, count:%d" % (item.key, item.row_count))
```

- Example 2

  The following code provides an example on how to group people aged 18 by score and obtain the two least common scores and the number of people with each score.

```
group_by = GroupByField('score', size = 2, group_by_sort = [RowCountSort(sort_order=SortO
rder.ASC)])
search_response = client.search(table_name, index_name,
                                SearchQuery(TermQuery('age', 18), limit=100, get_total_co
unt=True, group_bys = [group_by]),
                                ColumnsToGet(return_type=ColumnReturnType.ALL_FROM_INDEX)
)
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("key:%s, count:%d" % (item.key, item.row_count))
```

- Example 3

  The following code provides an example on how to group people aged 18 by score and obtain the top two common scores and the number of people in each group. Then, sort each group of people by primary key and obtain the information of the top three people in each group.

```
sort = RowCountSort(sort_order = SortOrder.DESC)
sub_agg = [TopRows(limit=3,sort=Sort([PrimaryKeySort(sort_order=SortOrder.DESC)]), name =
't1')]
group_by = GroupByField('l', size = 2, group_by_sort = [sort], sub_aggs = sub_agg)
search_response = client.search(table_name, index_name,
                                SearchQuery(TermQuery('age', 18), limit=100, get_total_co
unt=True, group_bys = [group_by]),
                                ColumnsToGet(return_type=ColumnReturnType.ALL_FROM_INDEX)
)
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("\tkey:%s, count:%d" % (item.key, item.row_count))
        for sub_agg in item.sub_aggs:
            print("\t\tname:%s:" % sub_agg.name)
            for entry in sub_agg.value:
                print("\t\t\tvalue:%s" % str(entry))
```

- Example 4

  The following code provides an example on how to group people aged 18 by score or gender:

```
sort = RowCountSort(sort_order = SortOrder.ASC)
sub_group = GroupByField('sex', size = 10, group_by_sort = [sort])
group_by = GroupByField('score', size = 10, group_by_sort = [sort], sub_group_bys = [sub_
group])
search_response = client.search(table_name, index_name,
                                SearchQuery(TermQuery('age', 18), limit=100, get_total_co
unt=True, group_bys = [group_by]),
                                ColumnsToGet(return_type=ColumnReturnType.ALL_FROM_INDEX)
)
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("\tkey:%s, count:%d" % (item.key, item.row_count))
        for sub_group in item.sub_group_bys:
            print("\t\tname:%s:" % sub_group.name)
            for sub_item in sub_group.items:
                print("\t\t\tkey:%s, count:%s" % (str(sub_item.key), str(sub_item.row_cou
nt)))
```

Group by range

The aggregation method that can be used to group query results based on the value ranges of a field. Field values that are within a specified range are grouped together. The number of values in each range is returned.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on the name.
field	The name of the field used to perform the aggregation operation. Only the LONG and DOUBLE data types are supported.
range[double_from, double_to)	The value ranges for grouping. The value range can start from Double.MIN_VALUE and end at Double.MAX_VALUE.
sub_aggs and sub_group_bys	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results. For example, after you group query results by sales volume and then by province, you can obtain the result of which province has the largest proportion in a specified range of sales volume. You must specify GroupByField in GroupByRange to run this query.

- Examples

  The following code provides an example on how to group people aged 18 by score range and obtain the numbers of people with scores in the [80, 90) and [90, 100) intervals:

```
query = TermQuery('age', 18)
group_by = GroupByRange(field_name = 'score', ranges = [(80, 90),(90, 100)])
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=0, group_bys
= [group_by]),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("range:%.1f~%.1f, count:%d" % (item.range_from, item.range_to, item.row_cou
nt))
```

Group by geographical distance

The aggregation method that can be used to group query results based on geographical locations to a central point. Query results in distances that are within a specified range are grouped together. The number of items in each range is returned.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on the name.
field	The name of the field used to perform the aggregation operation. Only the GEOPOINT data type is supported.
origin(double lat, double lon)	The longitude and latitude of the central point. lat indicates the latitude of the central point. lon indicates the longitude of the central point.
range[double_from, double_to)	The ranges for grouping. Unit: meter. The value range can start from Double.MIN_VALUE and end at Double.MAX_VALUE.
sub_aggs and sub_group_bys	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results.

- Examples

  The following code provides an example on how to group people aged 18 by the geographical distance from their school to their home locations and obtain the numbers of people who live within one kilometer from the school and people who live one to two kilometers away from school: The latitude and longitude of the school is (31,116).

```
query = TermQuery('age', 18)
group_by = GroupByGeoDistance(field_name = 'address', origin=GeoPoint(31, 116), ranges =
[(0, 1000), (1000,2000)])
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, group_bys
= [group_by]),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("range:%.1f~%.1f, count:%d" % (item.range_from, item.range_to, item.row_cou
nt))
```

Group by filter

The aggregation method that can be used to filter the query results and group them together to obtain the number of results that match each filter. Results are returned in the order in which the filters are specified.

- Parameters

Parameter	Description
name	The unique name specified for the aggregation operation. You can query results of the aggregation operation based on the name.
filter	The filters for the query. Results are returned in the order in which the filters are added.
sub_aggs and sub_group_bys	The sub-aggregation. The sub-aggregation operation is performed based on the grouping results.

- Examples

  The following code provides an example on how to group people aged 18 by filter and obtain the numbers of people who scored 100 in the math exam and who scored 100 in the Chinese exam.

```
query = TermQuery('age', 18)
filter1 = TermQuery('math', 100)
filter2 = TermQuery('chinese', 100)
filters = [filter1, filter2]
group_by = GroupByFilter(filters)
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, next_token = None, limit=2, group_bys = [group_by]),
    columns_to_get = ColumnsToGet(return_type = ColumnReturnType.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    i = 0
    for item in group_by.items:
        print("filter:%s=%s, count:%d" % (str(filters[i].field_name), str(filters[i].colu
mn_value), item.row_count))
        i=i+1
```

4.8. Secondary Index

4.8.1. Global secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created. The value of time_to_live is -1. The value of max_versions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table for an existing data table.

> **Note** You can call the CreateTable operation to create one or more index tables when you create a data table. For more information, see Create data tables.

- Parameters

Parameter	Description
main_table_name	The name of the data table.

Parameter	Description
index_meta	The schema information of the index table. The schema information contains the following items: ○ index_name: the name of the index table. ○ primary_key_names: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ○ defined_column_names: the attribute columns of the index table. The attribute columns are a combination of predefined columns of the data table. ○ index_type: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If index_type is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to index and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If index_type is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table.
include_base_data	Specifies whether to include the existing data from the data table in the index table. When the include_base_data parameter is set to true, the index table contains the existing data. When the value is set to false, the index table does not contain the existing data.

- Examples

  The following code provides an example on how to create an index table on a data table. The data table contains primary key columns pk1 and pk2. The index table contains primary key columns definedcol1 and pk1 and attribute columns definedcol2 and definedcol3.

```
index_meta = SecondaryIndexMeta('index2', ['definedcol1', 'pk1'], ['definedcol2', 'define
dcol3'])
client.create_secondary_index(table_name, index_meta)
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to be returned, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  When you call the GetRow operation to read data from an index table, take note of the following items:

  - You must set table_name to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from an index table

  For more information, see Multi-row operations.

  When you call the GetRange operation to read data from an index table, take note of the following items:

  - You must set table_name to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
main_table_name	The name of the data table.
index_name	The name of the index table.

- Examples

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

4.8.2. Local secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- Client is initialized. For more information, see Initialization.

- A data table is created. The value of time_to_live is -1. The value of max_versions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table for an existing data table.

> **Note** You can call the CreateTable operation to create one or more index tables when you create a data table. For more information, see Create data tables.

- Parameters

Parameter	Description
main_table_name	The name of the data table.
index_meta	The schema information of the index table. The schema information contains the following items: ○ index_name: the name of the index table. ○ primary_key_names: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ○ defined_column_names: the attribute columns of the index table. The attribute columns are a combination of predefined columns of the data table. ○ index_type: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If index_type is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to index and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If index_type is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table.

Parameter	Description
include_base_data	Specifies whether to include the existing data from the data table in the index table. When the include_base_data parameter is set to true, the index table contains the existing data. When the value is set to false, the index table does not contain the existing data.

- Examples

  The following code provides an example on how to create an index table. The data table contains the pk1 and pk2 primary key columns. The index table contains the pk1 and definedcol1 primary key columns and the definedcol2 and definedcol3 attribute columns.

```
index_meta = SecondaryIndexMeta('index2', ['pk1', 'definedcol1'], ['definedcol2', 'define
dcol3'],index_type= SecondaryIndexType.LOCAL_INDEX)
client.create_secondary_index(table_name, index_meta)
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to be returned, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  When you call the GetRow operation to read data from an index table, take note of the following items:

  - You must set table_name to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Read data within a specified range from an index table

  For more information, see Multi-row operations.

  When you call the GetRange operation to read data from an index table, take note of the following items:

  - You must set table_name to the name of the index table.
  - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
main_table_name	The name of the data table.
index_name	The name of the index table.

- Examples

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

4.9. Error handling

This topic describes the error handling methods and retry policies of Tablestore SDK for Python.

Methods

Tablestore SDK for Python handles errors as exceptions. If the operation succeeds, the called operation does not return an exception. If the operation fails, an exception is returned.

> **Note** Batch operations such as BatchGetRow and BatchWriteRow are successful only when the system verifies that no exception is returned and that the status of each row is successful.

Exceptions

Tablestore SDK for Python has two types of exceptions: OTSClientError and OTSServiceError. Both are inherited from Exception.

- OTSClientError: an internal SDK exception, such as an invalid parameter value or a failure to parse the results returned by the server.
- OTSServiceError: a server error. When a server error occurs, the server parses and returns the error message to the client. OTSServiceError has the following components:
  - get_http_status: an HTTP status code such as 200 and 404.
  - get_error_code: an error type string returned by Tablestore.
  - get_error_message: an error message string returned by Tablestore.
  - get_request_id: the UUID that identifies a request. If a problem persists, record the request ID and submit a ticket.

Retry

- Tablestore SDK for Python automatically retries operations in which an error occurs. In the default retry policy, the maximum retry attempts is 20, and the maximum retry interval is 3,000 ms. For more information about the retry policies for throttling errors and internal server errors related to write operations, see tablestore/retry.py.
- You can also customize a retry policy by inheriting the RetryPolicy class, and pass in the custom retry policy when you construct an OTSClient object.

Tablestore SDK for Python provides the following retry policies:

- DefaultRetryPolicy: the default retry policy. Only read operations are retried. The maximum number of retry attempts is 20, and the maximum retry interval is 3,000 ms.

- NoRetryPolicy: No operation is retried.
- NoDelayRetryPolicy: a retry policy without delay. Exercise caution when you configure this policy.
- WriteRetryPolicy: The write operations are retried based on the default retry policy.

5.Node.js SDK

5.1. Preface

This topic describes how to use Tablestore SDK for Node.js. The content in this topic applies to Tablestore SDK for Node.js V4.x.x.

Prerequisites

- Tablestore is activated. For more information, see Activate Tablestore.
- An AccessKey pair is created. For more information, see Obtain an AccessKey pair.

Download Tablestore SDK for Node.js

Download Tablestore SDK for Node.js from GitHub. For more information, visit GitHub.

For more information about the versions of Tablestore SDK for Python, see Tablestore SDK for Node.js.

Version

Latest version: V4.1.0.

5.2. Installation

This topic describes how to install Tablestore SDK for Node.js.

Prerequisites

Node.js 4.0 or later is installed.

> **Note** To avoid incompatibility problems, we recommend that you do not use Node.js versions 12.0 to 12.14.

Install Tablestore SDK for Node.js

Run the following command to install Tablestore SDK for Node.js:

```
npm install tablestore
```

Sample programs

Tablestore SDK for Node.js provides a variety of sample programs for your reference and use. You can obtain the sample programs by using one of the following methods:

- Download and decompress the Tablestore SDK for Node.js package. Find the sample programs in the examples folder.
- Access Tablestore SDK for Node.js on GitHub.

5.3. Initialization

Client is the client for Tablestore. Client provides a series of methods for you to manage tables and perform read and write operations on one or more rows.

Obtain an endpoint

An endpoint is a domain name that is used to access a Tablestore instance in a region. To query an endpoint, perform the following steps:

1. Log on to the Tablestore console.
2. Click the name of the instance to go to the **Instance Details** tab.

   On the Instance Details tab, you can view the endpoints of the instance in the Instance Access URL section.

> **Note** For more information about endpoints, see Endpoint.

Configure an AccessKey pair

To access Tablestore, you must have an AccessKey pair that consists of an AccessKey ID and AccessKey secret to verify your identity. The following types of AccessKey pairs are supported:

- The AccessKey pair of an Alibaba Cloud account. To obtain the AccessKey pair of an Alibaba Cloud account, take the following steps:
  i. On the Alibaba Cloud official website,Create Your Alibaba Cloud Account.
  ii. Create an AccessKey ID and an AccessKey secret. For more information, see Obtain an AccessKey pair.
- The AccessKey pair of a RAM user who is granted with permissions to access Tablestore. To obtain the AccessKey pair of a RAM user, take the following steps:
  i. Use your Alibaba Cloud account to log on to RAM. Create a RAM user or use an existing RAM user.
  ii. Use your Alibaba Cloud account to authorize the RAM user to access Tablestore.
  iii. After the RAM user is authorized, you can use the AccessKey pair of the RAM user to access Tablestore.
- Temporary access credentials obtained from STS. To obtain temporary access credentials from STS, take the following steps:
  i. The application server uses RAM or STS to obtain access credentials that consist of a temporary AccessKey ID, an AccessKey secret, and a token. After the access credentials are obtained, the application server sends them to you.
  ii. You can use the received access credentials to access Tablestore.

Initialize a Tablestore client instance

After you obtain the AccessKey ID and AccessKey secret, you can perform the following operations to initialize a Tablestore client instance:

```
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. Data type

This topic describes the correspondence between the five data types supported by Tablestore and the data types supported by Tablestore SDK for Node.js.

> **Note** For more information about the data types supported by Tablestore, see Naming conventions and data types.

Data type in Tablestore	Data type in SDK for Node.js	Description
String	string	The basic data type in JavaScript.
Integer	int64	The type of data encapsulated by SDK for Node.js.
Double	number	The basic data type in JavaScript.
Boolean	boolean	The basic data type in JavaScript.
Binary	Buffer	The Buffer object of Node.js.

The Integer data type in Tablestore is 64-bit signed integers and does not have a corresponding data type in JavaScript. A data type that can support 64-bit signed integers is required for Node.js. The following code provides an example on how to convert the integer data type in Tablestore:

```
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

5.5.1. Create data tables

This topic describes how to create a data table by calling the CreateTable operation. When you call the CreateTable operation, you must specify the schema information and configuration information of the data table when you call the CreateTable operation. You can set reserved read throughput and reserved write throughput for data tables in high-performance instances. You can create one or more index tables when you create a data table.

> **Note**
>
> - After a data table is created, it takes several seconds to load the data table. During this period, read and write operations performed on the table may fail. Perform operations on the data table after the data table is loaded.
> - You must specify the primary key when you create a data table. A primary key can contain one to four primary key columns. Each primary key column has a name and a data type.

Prerequisites

- An instance is created in the Tablestore console. For more information, see Create instances.
- Client is initialized. For more information, see Initialization.

Operations

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

Parameters

Parameter	Description

Parameter	Description
tableMeta	The schema information of the data table. The schema information contains the following items: • tableName: the name of the data table. • primaryKey: the schema of the primary key for the data table. For more information, see Primary keys and attributes. **Note** Attribute columns are optional. Different rows in Tablestore can have different attribute columns. You can specify the names of attribute columns when you write data to the attribute columns. ◦ The primary key of a data table can contain one to four primary key columns. Primary key columns are sorted in the order in which they are added. For example, PRIMARY KEY (A, B, C) and PRIMARY KEY (A, C, B) have different primary key schemas. Tablestore sorts rows based on the values of all primary key columns. ◦ The first primary key column serves as the partition key. Data that has the same partition key value is stored in the same partition. Therefore, we recommend that you assign no more than 10 GB of data to partitions that have the same partition key value. Otherwise, a single partition may be too large to split. We recommend that you evenly distribute requests to read and write data to partitions that have different partition keys for load balancing. • definedColumn: the predefined columns of the table and the data types of the predefined column values. Primary key columns cannot be set to predefined columns. You can use predefined columns as the indexed columns or attribute columns for index tables.

Parameter	Description
tableOptions	The configuration information of the data table. For more information, see Data versions and TTL. The configurations include the following items: • timeToLive: the period for which data in the table can be retained. This period is the validity period of data. When the retention period exceeds the timeToLive value, Tablestore automatically deletes expired data. The minimum timeToLive value is 86400, which is equal to one day. A value of -1 specifies that data never expires. When you create a data table, you can set timeToLive to -1 if you expect that data never expires. After the table is created, you can call the UpdateTable operation to modify the timeToLive value. Unit: seconds. **Note** You must set timeToLive to -1 if you want to create an index table for a data table. • maxVersions: the maximum number of versions of data that can be retained for a single attribute column. When the number of versions of data in attribute columns exceeds the maxVersions value, Tablestore automatically deletes the data of earlier versions. When you create a data table, you can customize the maximum number of versions that can be retained for the data in an attribute column. After the data table is created, you can call the UpdateTable operation to modify the maxVersions value for the data table. **Note** You must set maxVersions to 1 if you want to create an index table for a data table.

Parameter	Description
reservedThroughput	The reserved read throughput and reserved write throughput of the data table. The reserved read throughput and reserved write throughput can be set to 0 for data tables only in capacity instances. Reserved throughput does not apply to these instances. The default value 0 specifies that all throughput is billed on a pay-as-you-go basis. Unit: capacity unit (CU). • If reserved read throughput or reserved write throughput is set to a value greater than 0 for a data table, Tablestore allocates and reserves related resources for the data table. After the data table is created, Tablestore charges reserved throughput resources. Additional throughput is billed on a pay-as-you-go basis. For more information, see Billing overview. • If reserved read throughput or reserved write throughput is set to 0, Tablestore does not allocate or reserve related resources for the data table.

Parameter	Description
indexMetas	The schema information of the index tables. Each indexMeta contains the following items: • name: the name of the index table. • primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. • definedColumn: the indexed attribute columns. The attribute columns are a combination of predefined columns of the data table. • includeBaseData: specifies whether to include the existing data from the data table in the index table. When the includeBaseData parameter in CreateIndex is set to true, the index table includes the existing data. When the value is set to false, the index table excludes the existing data. • indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ◦ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to index and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ◦ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. • indexUpdateMode: the update mode of the index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ◦ If indexUpdateMode is not specified or is set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the global secondary index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ◦ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the index update mode to IUM_SYNC_INDEX.

Examples

- Create a data table without creating an index table

  The following code provides an example on how to create a data table that has two primary key columns and reserved read and 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, // Specify the validity period of data. A value of -1 specifies that the data never expires. Unit: seconds. If you set the validity period to a year, the value of timeToLive is 31,536,000 (365 × 24 × 3600).
    maxVersions: 1  // Specify the maximum number of versions that can be retained for each column. A value of 1 specifies that only the latest version is retained for each column.
  }
};
client.createTable(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

  For the detailed sample code, visit CreateTable@GitHub.

- Create a data table and an index table whose index type is global secondary index

```
var client = require('./client');
var TableStore = require('../index.js');
var params = {
  tableMeta: {
    tableName: 'sdkGlobalTest',
    primaryKey: [
```

```
      {
        name: 'pk1',
        type: TableStore.PrimaryKeyType.INTEGER
      },
      {
        name: 'pk2',
        type: TableStore.PrimaryKeyType.INTEGER
      }
    ],
    definedColumn: [
      {
        "name": "col1",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      },
      {
        "name": "col2",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      }
    ],
  },
  reservedThroughput: {
    capacityUnit: {
      read: 0,
      write: 0
    }
  },
  tableOptions: {
    timeToLive: -1, // Specify the validity period of data. A value of -1 specifies that
the data never expires. Unit: seconds. You must set timeToLive to -1 for a data table tha
t has index tables.
    maxVersions: 1  // Specify the maximum number of versions that can be retained for ea
ch column. A value of 1 specifies that only the latest version is retained for each colum
n. You must set maxVersions to 1 for a data table that has index tables.
  },
  streamSpecification: {
    enableStream: false,// The stream feature cannot be enabled for secondary indexes.
  },
  indexMetas: [
    {
      name: "sdkGlobalIndex1",
      primaryKey: ["pk2"],
      definedColumn: ["col1", "col2"]
    },
    {
      name: "sdkGlobalIndex2",
      primaryKey: ["col1"],
      definedColumn: ["col2"]
    }
  ]
};
client.createTable(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
```

```
  }
  console.log('success:', data);
});
```

- Create a data table and an index table whose index type is local secondary index

```
var client = require('./client');
var TableStore = require('../index.js');
var params = {
  tableMeta: {
    tableName: 'sdkLocalTest',
    primaryKey: [
      {
        name: 'pk1',
        type: TableStore.PrimaryKeyType.INTEGER
      },
      {
        name: 'pk2',
        type: TableStore.PrimaryKeyType.INTEGER
      }
    ],
    definedColumn: [
      {
        "name": "col1",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      },
      {
        "name": "col2",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      }
    ],
  },
  reservedThroughput: {
    capacityUnit: {
      read: 0,
      write: 0
    }
  },
  tableOptions: {
    timeToLive: -1, // Specify the validity period of data. A value of -1 specifies that the data never expires. Unit: seconds. You must set timeToLive to -1 for a data table that has index tables.
    maxVersions: 1  // Specify the maximum number of versions that can be retained for each column. A value of 1 specifies that only the latest version is retained for each column. You must set maxVersions to 1 for a data table that has index tables.
  },
  streamSpecification: {
    enableStream: false,// The stream feature cannot be enabled for secondary indexes.
  },
  indexMetas: [
    {
      name: "sdklocalIndex1",
      primaryKey: ["pk1","col1"],// Add primary key columns to the index table. The first primary key column of the index table must be the same as the first primary key column of the data table
```

```
the data table.
      definedColumn: ["col2"],
      indexUpdateMode: TableStore.IndexUpdateMode.IUM_SYNC_INDEX,// Set the index update
mode to IUM_SYNC_INDEX, which specifies the synchronous update mode. If the index type is
set to IT_LOCAL_INDEX, the index update mode must be set to IUM_SYNC_INDEX.
      indexType: TableStore.IndexType.IT_LOCAL_INDEX,// Set the index type to IT_LOCAL_IN
DEX, which specifies a local secondary index.
    },
    {
      name: "sdklocalIndex2",
      primaryKey: ["pk1","col2"],
      definedColumn: ["col1"],
      indexUpdateMode: TableStore.IndexUpdateMode.IUM_SYNC_INDEX,// Set the index update
mode to IUM_SYNC_INDEX, which specifies the synchronous update mode. If the index type is
set to IT_LOCAL_INDEX, the index update mode must be set to IUM_SYNC_INDEX.
      indexType: TableStore.IndexType.IT_LOCAL_INDEX,// Set the index type to IT_LOCAL_IN
DEX, which specifies a local secondary index.
    }
  ]
};
client.createTable(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

5.5.2. UpdateTable

You can call this operation to update the maximum number of versions, reserved read throughput, or reserved write throughput of the specified table.

Operations

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

Examples

The following code provides an example on how to update the maximum number of versions of a table to five:

```
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);
});
```

For the detailed sample code, visit UpdateTable@GitHub.

5.5.3. List the names of tables

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

> **Note** For more information about the ListTable operation, see ListTable.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created.

API operations

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

Examples

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

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

For the detailed sample code, visit ListTable@GitHub.

5.5.4. Query the description of a table

> **Note** For more information about the DescribeTable operation, see DescribeTable.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created.

API operations

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

Parameters

Parameter	Description
tableName	The name of the table.

Examples

The following code provides an example on how to query the description of a table:

```
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);
});
```

For the detailed sample code, visit DescribeTable@GitHub.

5.5.5. Delete tables

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

Prerequisites

- Client is initialized. For more information, see Initialization.

- A table is created.
- The index tables and search indexes for the table are deleted.

Operations

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

Parameters

Parameter	Description
tableName	The name of the table.

Examples

The following code provides an example on how to delete a specified table:

```
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);
});
```

For the detailed sample code, visit DeleteTable@GitHub.

5.5.6. Configure an auto-increment primary key column

This topic describes how to configure an auto-increment primary key column. You cannot set a partition key to an auto-increment column. If you write data to a table that contains an auto-increment primary key column, you do not need to specify specific values for the auto-increment primary key column because Tablestore generates values for the auto-increment primary key column. Values generated for the auto-increment primary key column are unique and consecutive within the partition that shares the same partition key value.

Prerequisites

The OTSClient instance is initialized. For more information, see Initialization.

Usage notes

1. When you create a table, you cannot set a partition key to an auto-increment primary key column.

   The data type of an auto-increment primary key column can only be set to integer. Each value generated for an auto-increment primary key column is a 64-bit signed long integer.

2. When you write data to a table, you do not need to specify a specific value for the auto-increment column. You need only to set a placeholder value for the auto-increment column.

   If you want to obtain the value of the auto-increment column after data is written to the table, you can set ReturnType to Primarykey.

   When you query data, you must specify the values of all primary key columns. To obtain a complete primary key value, you can set ReturnType to Primarykey in PutRow, UpdateRow, or BatchWriteRow.

Examples

When you use auto-increment primary key columns, you can call the CreateTable, PutRow, UpdateRow, or BatchWriteRow operation.

1. Create a table

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

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var tableName = "autoIncTable";
var pk1 = "stringPK";
var pk2 = "autoIncPK";
function createTableWithAutoIncrementPk() {
    var createParams = {
        tableMeta: {
            tableName: tableName,
            primaryKey: [
                {
                    name: pk1,
                    type: 'STRING'
                },
                {
                    name: pk2,
                    type: 'INTEGER',
                    option: 'AUTO_INCREMENT'// Set the auto-increment column by setting
option to AUTO_INCREMENT.
                },
            ]
        },
        reservedThroughput: {
            capacityUnit: {
                read: 0,
                write: 0
            }
        },
        tableOptions: {
            timeToLive: -1,// Specify the validity period of data in seconds. A value o
f -1 indicates that the data never expires. If you set this parameter to 365×24*3600, t
he validity period of the data is one year.
            maxVersions: 1// Specify the maximum number of versions that can be saved i
n each column. A value of 1 indicates that only the latest version is saved in each col
umn.
        },
    };
    client.createTable(createParams, function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('create table success');
    });
}
```

2. Write data

   When you write data to a table, you do not need to specify a value of the auto-increment column. You need only to set a placerholder value for the column.

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var tableName = "autoIncTable";
var pk1 = "stringPK";
var pk2 = "autoIncPK";
function putRow() {
    var putParams = {
        tableName: tableName,
        condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE,
null),
        primaryKey: [
            { stringPK: 'pk1' },
            { autoIncPK: TableStore.PK_AUTO_INCR }
        ],
        attributeColumns: [
            { 'col1': 'col1val' }
        ],
        returnContent: { returnType: TableStore.ReturnType.Primarykey }
    };
    client.putRow(putParams, function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('put row success,autoIncrement pk value:' + JSON.stringify(data.row
.primaryKey));
    });
}
```

5.5.7. Configure conditional update

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

When you use the PutRow, UpdateRow, DeleteRow, or BatchWriteRow operation to update data, you can use conditional update to check row existence conditions and column-based conditions. The update is successful only when the conditions are met.

Conditional update includes row existence conditions and column-based conditions.

- Row existence conditions: include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.

  When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported.

- Column-based conditions: include SingleColumnCondition and CompositeCondition, which are used to determine whether the conditions are met based on the values of one or more columns.

- SingleColumnCondition supports the comparison between a constant and a column that can be a primary key column. SingleColumnCondition does not support the comparison between two columns or two constants.
- The inner node of CompositeCondition is used to perform logical operations. Subconditions can be SingleColumnCondition or CompositeCondition.

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

Parameters

Parameter	Description
RowExistenceExpectation	When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported. Row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST. In Tablestore, RowExistenceExpectation_IGNORE indicates IGNORE, RowExistenceExpectation_EXPECT_EXIST indicates EXPECT_EXIST, and RowExistenceExpectation_EXPECT_NOT_EXIST indicates EXPECT_NOT_EXIST. • IGNORE: No existence check is performed. • EXPECT_EXIST: The row is expected to exist. If the row exists, the condition is met. If the row does not exist, the condition is not met. • EXPECT_NOT_EXIST: Expect that the row does not exist. If the row does not exist, the condition is met. If the row exists, the condition is not met.
columnName	The column name.
columnValue	The comparison value of the column.
comparator	The relational operator that is used to compare column values. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=). In Tablestore, CT_EQUAL indicates EQUAL (=), CT_NOT_EQUAL indicates NOT_EQUAL (!=), CT_GREATER_THAN indicates GREATER_THAN (>), CT_GREATER_EQUAL indicates GREATER_EQUAL (>=), CT_LESS_THAN indicates LESS_THAN (<), and CT_LESS_EQUAL indicates LESS_EQUAL (<=).
combinator	The logical operator that is used to combine multiple conditions. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. In Tablestore, LO_NOT indicates NOT, LO_AND indicates AND, and LO_OR indicates OR. Different logical operators require different number of subconditions. • When the logical operator is NOT, only one subcondition can be added. • When the logical operator is AND or OR, you must add at least two subconditions.

Parameter	Description
filterIfMissing	Specifies whether to pass the conditional check when a column does not exist in a row. The type of this parameter value is Boolean. The default value is true, which indicates that if the column does not exist in a row, the conditional check is passed, and the row meets the update conditions. When filterIfMissing is set to false, if the column does not exist in a row, the conditional check fails, and the row does not meet the update conditions.
latestVersionOnly	Specifies whether to use only the value of the latest version when the comparison value has multiple versions. The type of this parameter value is Boolean. The default value is true, which indicates that if the column has multiple versions of data, only the value of the latest version is used for comparison. When latestVersionOnly is set to false, the comparison values of all versions are used for comparison if the comparison value has multiple versions. In this case, if only the value of a version meets the condition, the conditional check is passed, and the row meets the update conditions.

Examples

Update a row based on the specified primary key. If the specified row exists, the value of the name column is 'john', and the value of the addr column is 'china', the update succeeds. Otherwise, the update fails.

```
   var params = {
    tableName: "sampleTable",
    primaryKey: [{ 'gid': Long.fromNumber(20013) }, { 'uid': Long.fromNumber(20013) }],
    updateOfAttributeColumns: [{ 'PUT': [{ 'col1': 'test6' }] }]
};
// Specify that the expected row exists, that the value of the name column is 'john', and t
hat the value of the addr column is 'china'.
var condition = new TableStore.CompositeCondition(TableStore.LogicalOperator.AND);
condition.addSubCondition(new TableStore.SingleColumnCondition('name', 'john', TableStore.C
omparatorType.EQUAL));
condition.addSubCondition(new TableStore.SingleColumnCondition('addr', 'china', TableStore.
ComparatorType.EQUAL));
params.condition = new TableStore.Condition(TableStore.RowExistenceExpectation.EXPECT_EXIST
, condition);
client.updateRow(params,
    function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('success:', data);
    });
```

5.5.8. Configure local transaction

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

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

You can use local transaction to specify that the operations on a partition key are atomic, which indicates that operations based on the specified partition key either succeed or fail. The isolation level of the local transaction is read committed.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

1. Use startLocalTransaction to create a local transaction based on the specified partition key value and obtain the ID of the local transaction.
2. Read and write data within the local transaction.

   You can call the following operations on the local transaction: GetRow, PutRow, DeleteRow, UpdateRow, BatchWriteRow, and GetRange.
3. Use commitTransaction to commit local transactions or use abortTransaction to abort local transactions.

Limits

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

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

    If a transaction is not read or written for more than 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.
  - Up to 4 MB of data can be written to each transaction. The volume of data written to each transaction is calculated in the same way as a regular write request.

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

Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the table. • When you create a local transaction, you need only to specify the partition key value for the local transaction. • After a local transaction is created, you must specify a complete primary key when you read data from or write data to the local transaction.
transactionId	The ID that identifies a local transaction. After a local transaction is created, the transaction ID must be included for operations on the local transaction.

Examples

- Commit a local transaction for all modified data to take effect.

```
(async () => {
    try {
        // Create a local transaction.
        const response = await client.startLocalTransaction({
            tableName,
            primaryKey: [{  // You need only to specify the partition key value for the l
ocal transaction.
                "id": "partitionKeyValue"
            }]
        });
        // Obtain the local transaction ID.
        const transactionId = response.transactionId;
        // Write data within the local transaction.
        await client.putRow({
            tableName,
            condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE
, null),
            primaryKey,
            attributeColumns: [{
                col: 'updated'
            }],
            transactionId
        });
        // Commit the local transaction.
        await client.commitTransaction({
            transactionId
        })
    } catch (e) {
        console.error(e)
    }
})();
```

- Abort a local transaction to ignore all modified data.

```
(async () => {
    try {
        // Create a local transaction.
        const response = await client.startLocalTransaction({
            tableName,
            primaryKey: [{  // You need only to specify the partition key value for the l
ocal transaction.
                "id": "partitionKeyValue"
            }]
        });
        // Obtain the local transaction ID.
        const transactionId = response.transactionId
        // Write data within the local transaction.
        await client.putRow({
            tableName,
            condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE
, null),
            primaryKey,
            attributeColumns: [{
                col: 'updated'
            }],
            transactionId
        });
        // Abort the local transaction.
        await client.abortTransaction({
            transactionId
    })
  } catch (e) {
    console.error(e)
  }
})();
```

5.5.9. Configure atomic counter

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

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Limits

- You can implement atomic counters only on INTEGER columns.
- By default, if a column that is specified as an atomic counter does not exist, the value of the column is 0 before you write data. If a column that is specified as an atomic counter is not an INTEGER column, an OTSParameterInvalid error occurs.
- You can update an atomic counter by using a positive or negative number, but you must avoid an integer overflow. If an integer overflow occurs, an OTSParameterInvalid error is returned.
- By default, the value of an atomic counter is not returned in the response to an update row request. You can specify that the increased value of an atomic counter is returned.

- You cannot specify a column as an atomic counter and update the column in a single request. If Attribute Column A is set to an atomic counter, you cannot perform other operations such as overwrite and delete operations on the attribute column A.
- You can perform multiple update operations on the same row by using a BatchWriteRow request. However, if you perform an atomic counter operation on a row, you can perform only one update operation on the row in a BatchWriteRow request.
- Only the value of the latest version of an atomic counter is increased. You cannot increase the value of a specified version of an atomic counter. After you update a row, a new version of data is inserted to the atomic counter in the row.

API operations

The API operations to implement the atomic counter feature are added to the updateRow operation. The following table lists these operations.

Operation	Description
updateOfAttributeColumns	Increases or decreases a value by a number. The update type is INCREMENT.
returnContent	Specifies the column name and return type on which operations are performed to implement atomic counter. The updated value in the column on which operations are performed to implement atomic counter must be returned.

Parameters

Parameter	Description
tableName	The name of the table.
columnName	The name of the column on which operations are performed to implement atomic counter. Columns whose valid values are integers are supported to implement the atomic counter feature.
value	The value to update the columns on which operations are performed to implement atomic counter.
returnColumns	Specifies the column names whose value needs to be returned for the columns on which operations are performed to implement atomic counter.
returnType	Sets the return type to TableStore.ReturnType.AfterModify and returns the values of the columns on which operations are performed to implement atomic counter.

Examples

The following code provides an example on how to call the updateRow operation to update the columns whose valid values are integers when you write a row of data. Set the type to INCREMENT for the attribute column:

```
var params = {
  tableName: "<Your-Table-Name>",
  condition: new TableStore.Condition(TableStore.RowExistenceExpectation.EXPECT_EXIST, null
),
  primaryKey: [{'pk0': Long.fromNumber(1)}],
  // Specify that the value of the price column on which operations are performed to implem
ent atomic counter is increased by 10. Do not set timestamps.
  updateOfAttributeColumns: [
    {'INCREMENT': [{'price': Long.fromNumber(10)}]}
  ],
  // Set the return type to TableStore.ReturnType.AfterModify and return the values of the
columns on which operations are performed to implement atomic counter.
  returnContent: {
    returnColumns: ["price"],
    returnType: TableStore.ReturnType.AfterModify
  }
};
client.updateRow(params,
  function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
  });
```

5.5.10. Configure a filter

Tablestore filters query results on the server side before returning the results. Only rows of data that match the filter conditions are returned.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created and data is written to the table.

Usage notes

When you call the GetRow, BatchGetRow, or GetRange operation to query data, you can use a filter to return only the rows that meet the filter conditions.

Filters include SingleColumnCondition and CompositeColumnCondition.

- SingleColumnCondition: determines whether to filter a row based only on the values of a reference column.
- CompositeColumnCondition: determines whether to filter a row based on a combination of the filter conditions on values of multiple reference columns.

Limits

- A filter can consist of up to 10 conditions. Supported conditions include relational operations (=, !=, >, >=, <, and <=) and logical operations (NOT, AND, and OR).
- The reference columns used by a filter must be included in the read data. If you specify columns from

which data is read that do not include the reference columns, the filters cannot obtain the reference column values.

- The native semantics or limits of the GetRow, BatchGetRow, and GetRange operations are not affected when you use filters.

  You can scan up to 5,000 rows and return up to 4 MB of data with a single GetRange operation.

  If no data matches the filter conditions in the scanned range, the returned rows are empty. However, next_start_primary_key may not be empty. In this case, you must use next_start_primary_key to continue reading the data until next_start_primary_key is empty.

Parameters

Parameter	Description
columnName	The name of the reference column used by the filter.
columnValue	The value of the reference column used by the filter.
ComparatorType	The relational operator used by the filter. For more information, see ComparatorType. The relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUA L(<=).
LogicOperator	The logical operator used by the filter. For more information, see LogicalOperator. The logical operators include NOT, AND, and OR.
passIfMissing	Specifies whether to return a row when a reference column in the row does not exist. The data type of the parameter value is Boolean. The default value is true, which indicates that if a reference column does not exist in a row, the row is returned. If you set passIfMissing to false, a row is not returned if a reference column does not exist in the row.
latestVersionOnly	Specifies whether to use only the latest versions of data in the reference columns for comparison when each reference column contains data of multiple versions. The data type of the parameter value is Boolean. The default value is true, which indicates that the latest versions of data are used for comparison when each reference column contains data of multiple versions. If you set latestVersionOnly to false, all versions of data in a reference column are used for comparison. The row is returned if one version of data in the reference column meets the condition.

Examples

- The following code provides an example on how to construct a SingleColumnCondition:

```
function getRowWithCondition() {
  // Specify that the row is returned when the value of the col1 column is Tablestore. Wh
en passIfMissing is set to true, the row is returned if this column does not exist. When
passIfMissing is set to false, the row is not returned if this column does not exist.
  var condition = new TableStore.SingleColumnCondition('col1', 'Tablestore', TableStore.C
omparatorType.EQUAL,true);
  params.columnFilter = condition;
  client.getRow(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    console.log('success:', data);
  });
}
```

- The following code provides an example on how to construct a CompositeCondition:

```
function getRowWithCompositeCondition() {
  // Specify that the row is returned when the value of the col1 column is Tablestore, an
d the value of the col5 column is 123456789.
  var condition = new TableStore.CompositeCondition(TableStore.LogicalOperator.AND);
  condition.addSubCondition(new TableStore.SingleColumnCondition('col1', 'Tablestore', Ta
bleStore.ComparatorType.EQUAL));
  condition.addSubCondition(new TableStore.SingleColumnCondition('col5', Long.fromNumber(
123456789), TableStore.ComparatorType.EQUAL));
  params.columnFilter = condition;
  client.getRow(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    console.log('success:', data);
  });
}
```

5.6. Single-row operations

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

PutRow

You can call this operation to insert a row. If the row exists, data of all versions in all columns of the row is deleted before new data is inserted.

- Operations

```
  /**
   * Insert data into a specified row. If the row exists, the existing data is overwritte
n. Otherwise, a new row is added.
   */
  putRow(params, callback)
```

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** ◦ The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table. ◦ For an auto-increment primary key column, you need only to set the value of the auto-increment primary key column to a placeholder. For more information, see Configure an auto-increment primary key column.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update. **Note** ◦ RowExistenceExpectation.IGNORE indicates that new data is inserted into a row regardless of whether the specified row exists or not. If the specified row exists, the existing data is overwritten. ◦ RowExistenceExpectation.EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten. ◦ RowExistenceExpectation.EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist.

Parameter	Description
attributeColumns	The attribute columns of the row. ◦ Each attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value type (optional), attribute column value, and timestamp (optional). ◦ A timestamp is the data version number. For more information, see Data versions and TTL. You can customize a data version number or use the data version number generated by Tablestore. If you do not specify this parameter, the data version number generated by the system is used. ■ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ■ When you choose to specify the version number, ensure that the version number is a 64-bit timestamp accurate to the millisecond within the valid version range.
returnContent	The return type. returnType: Set the value to TableStore.ReturnType.Primarykey, which returns the primary key values. This parameter is used for auto-increment primary key columns.

- Examples

  The following code provides an example on how to insert a row of data:

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var currentTimeStamp = 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': 'Tablestore' },
    { 'col2': '2', 'timestamp': currentTimeStamp },
    { '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);
});
```

For the detailed sample code, visit PutRow@GitHub.

GetRow

You can call this operation to read a row of data.

The following results for the read request may be returned:

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

```
  /**
   * Read a single row of data based on a specified primary key.
   */
  getRow(params, callback)
```

- Parameters

Parameter	Description
tableName	The name of the table.

Parameter	Description
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
columnsToGet	The names of the columns to read. You can specify the names of primary key columns and the names of attribute columns. If you do not specify this parameter, all data in the row is returned. **Note** ◦ By default, Tablestore returns the data from all columns of the row when you query a row. You can use the columnsToGet parameter to read data from specified columns. For example, Tablestore returns only the values of col0 and col1 if col0 and col1 are included in columnsToGet. ◦ If columnsToGet and columnFilter are used at the same time, the system first obtains the columns specified by columnsToGet and then filters the returned columns.
maxVersions	The maximum number of versions that can be read. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data whose versions are within the specified time range or data of the specified version is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
timeRange	Specifies a range of versions to read or a version of data to read. For more information, see TimeRange. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data whose versions are within the specified time range or data of the specified version is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ To query data within a range, you must set startTime and endTime. startTime specifies the start timestamp. endTime specifies the end timestamp. The time range is a left-closed and right-open interval, which is [startTime, endTime). ◦ To query data within a range of versions, set specificTime. specificTime specifies a specific timestamp. Only one of specificTime and [startTime, endTime) is required. Valid values: [0, Long.MAX_VALUE). Unit: milliseconds.
columnFilter	You can set filter conditions to filter the queried results on the server side. Only rows that meet the specified filter conditions are returned. For more information, see Configure a filter. **Note** If columnsToGet and columnFilter are used at the same time, the system first obtains the columns specified by columnsToGet and then filters the returned columns.

- Examples

  The following code provides an example on how to read a row of data:

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sampleTable",
  primaryKey: [{ 'gid': Long.fromNumber(20004) }, { 'uid': Long.fromNumber(20004) }],
  maxVersions: 2 // Specify the maximum number of versions that can be read. A value of 2
indicates that you can read a maximum of two versions of data.
};
var condition = new TableStore.CompositeCondition(TableStore.LogicalOperator.AND);
condition.addSubCondition(new TableStore.SingleColumnCondition('name', 'john', TableStore
.ComparatorType.EQUAL));
condition.addSubCondition(new TableStore.SingleColumnCondition('addr', 'china', TableStor
e.ComparatorType.EQUAL));
params.columnFilter = condition;
client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

For the detailed sample code, visit GetRow@GitHub.

UpdateRow

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

> **Note** If the UpdateRow request contains only columns to delete, and the specified row does not exist, a new row is not added.

- Operations

```
  /**
   * 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 ba
sed on the request content.
   */
  updateRow(params, callback)
```

- Parameters

Parameter	Description
tableName	The name of the table.

Parameter	Description
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.
updateOfAttributeColumns	The attribute columns you want to update.

- Examples

  The following code provides an example on how to update the data of a specified row:

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
    tableName: "sampleTable",
    condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE, null),
    primaryKey: [{ 'gid': Long.fromNumber(9) }, { 'uid': Long.fromNumber(90) }],
    updateOfAttributeColumns: [
        { 'PUT': [{ 'col4': Long.fromNumber(4) }, { 'col5': '5' }, { 'col6': Long.fromNum
ber(6) }] },
        { 'DELETE': [{ 'col1': Long.fromNumber(1496826473186) }] },
        { 'DELETE_ALL': ['col2'] }
    ]
};
client.updateRow(params,
    function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('success:', data);
    });
```

  For the detailed sample code, visit UpdateRow@GitHub.

DeleteRow

You can call this operation to delete a row. If the specified row does not exist, no change is made to the table.

- Operations

```
/**
 * Delete a row of data.
 */
deleteRow(params, callback)
```

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.

- Examples

  The following code provides an example on how to delete a row from a table:

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
    tableName: "sampleTable",
    condition: new TableStore.Condition(TableStore.RowExistenceExpectation.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);
});
```

  For the detailed sample code, visit DeleteRow@GitHub.

5.7. Multi-row operations

BatchWriteRow

You can call this operation to write several rows of data to one or more tables by sending one request. The BatchWriteRow operation consists of multiple PutRow, UpdateRow, and DeleteRow operations. The process of constructing a suboperation is the same as that of calling the PutRow, UpdateRow, and DeleteRow operations. BatchWriteRow supports conditional updates.

The execution and returning of results of each suboperation are carried out independently.

When multiple rows are written, some rows may fail to be written. If the operation fails for some rows, an exception may not occur, but information about the index and error messages of the failed rows is stored in BatchWriteRowResponse. Therefore, when you call the BatchWriteRow operation, you must check the return values and check whether the status of each row is successful. If you do not check the return values, failures of operations on part of rows are ignored.

When the server detects errors caused by operations, the BatchWriteRow operation may return exceptions of parameter errors. All operations for the request are not executed.

- Operations

```
/**
 * Modify multiple rows of data.
 */
batchWriteRow(params, callback)
```

- Parameters

  BatchWriteRow is a combination of PutRow, UpdateRow, and DeleteRow.

  - Hierarchies are created for tables. Multiple tables can be processed at a time.

    The values of the tables parameter are based on tables. This parameter specifies information about tables and rows on which the write, modification, or delete operations are performed. For more information about parameters, see Single-row operations.

  - The type parameter is added to distinguish between different operation types.

    You can set the operation type to PUT, UPDATE, or DELETE.

    - If you set the operation type to PUT or UPDATE, primaryKey and attributeColumns are valid.
    - If you set the operation type to DELETE, primaryKey is valid.

- Examples

  The following code provides an example on how to write multiple rows of data:

```
var client = require('./client');
var TableStore = require('../index.js');
var Long = TableStore.Long;
var params = {
    tables: [
        {
            tableName: 'sampleTable',
            rows: [
                {
                    type: 'UPDATE',
                    condition: new TableStore.Condition(TableStore.RowExistenceExpectatio
n.IGNORE, null),
                    primaryKey: [{ 'gid': Long.fromNumber(20010) }, { 'uid': Long.fromNum
ber(20010) }],
                    attributeColumns: [{ 'PUT': [{ 'col1': 'test3' }, { 'col2': 'test4' }
] }],
                    returnContent: { returnType: 1 }
                },
                {
                    type: 'PUT',
                    condition: new TableStore.Condition(TableStore.RowExistenceExpectatio
n.IGNORE, null),
                    primaryKey: [{ 'gid': Long.fromNumber(20020) }, { 'uid': Long.fromNum
ber(20020) }],
                    attributeColumns: [{ 'col1': 'test1' }, { 'col2': 'test2' }],
                    returnContent: { returnType: TableStore.ReturnType.Primarykey }
                },
                {
                    type: 'DELETE',
                    condition: new TableStore.Condition(TableStore.RowExistenceExpectatio
n.IGNORE, null),
                    primaryKey: [{ 'gid': Long.fromNumber(20018) }, { 'uid': Long.fromNum
ber(20018) }],
                }
            ]
        }
    ],
};
client.batchWriteRow(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

For the detailed sample code, visit BatchWriteRow@GitHub.

BatchGetRow

You can call this operation to read several rows of data from one or more tables by sending one request. The BatchGetRow operation consists of multiple GetRow operations. The process of constructing a suboperation is the same as that of calling the GetRow operation. BatchGetRow supports filters.

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

The execution and returning of results of each suboperation are carried out independently.

When multiple rows are read, part of rows may fail to be read. If the operation fails for part of rows, the system does not return exceptions. However, information about the error messages of the failed rows is stored in BatchGetRowResponse. Therefore, when you call the BatchGetRow operation, you must check the return values and check whether the status of each row is successful.

- Operations

```
/**
 * Read multiple rows of data from one or more tables.
 */
batchGetRow(params, callback)
```

- Parameters

  Difference between BatchGetRow and GetRow:

  - Hierarchies are created for tables. Multiple tables can be read at a time.

    The values of the tables parameter are based on tables. This parameter specifies information about tables and rows on which various operations are performed. For more information about parameters, see Single-row operations.

  - The primaryKey parameter allows you to set primary keys for multiple rows. You can use the primaryKey parameter to read data from multiple rows at a time.

- Examples

  The following code provides an example on how to read multiple rows of data from multiple tables, and retry the operation if an error occurs during the read of 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);
```

For the detailed sample code, visit BatchGetRow@GitHub.

GetRange

You can call this operation to read rows of data in a forward or backward direction based on a specified primary key range.

You can also set the number of rows you want 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 the next primary key information are returned. You can also initiate a request to start from where the last operation left off and read the remaining rows based on the next primary key information returned by the previous operation.

The GetRange operation may stop and return data in the following situations:

- The amount of data read reaches 4 MB.
- The number of rows read reaches 5,000.
- The number of rows returned reaches the limit.
- All reserved read throughput is consumed. You have no reserved read throughput to read the next row of data.

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

- Operations

```
/**
 * Read data within the specified primary key range.
 */
getRange(params, callback)
```

- Parameters

Parameter	Description
tableName	The name of the table.
direction	The direction for reading. ◦ If the direction is set to FORWARD, the start primary key column value must be smaller than the end primary key column value. The rows are returned based on the primary key column values in ascending order. ◦ If the direction is set to BACKWARD, the start primary key column value must be greater than the end primary key column value. The rows are returned based on the primary key column values in descending order. Example: A table has two primary keys A and B where the value of A is smaller than that of B. If you set direction to forward, the rows whose primary key column values are greater than or equal to the value of A and smaller than the value of B are returned in ascending order. If you set direction to backward, the rows whose primary key column values are greater than the value of A and smaller than or equal to the value of B are returned in descending order.

Parameter	Description
inclusiveStartPrimaryKey exclusiveEndPrimaryKey	The start and end primary keys of the read operation. The start and end primary keys must be valid primary keys or virtual points of the INF_MIN or INF_MAX type. The number of columns for the virtual point must be the same as that of the primary key. INF_MIN indicates an infinitely small value of which all other values of other types are greater. INF_MAX indicates an infinitely large value of which all other values are smaller. ◦ inclusiveStartPrimaryKey specifies the start primary key. If the row exists, the response contains this row. ◦ exclusiveEndPrimaryKey specifies the end primary key. The response excludes the row no matter whether the row exists. Rows in tables are sorted based on the primary key column values in ascending order. The range for reading is a left-closed and right-open interval. When data is read in the forward direction, all rows whose primary key column values are greater than or equal to the start primary key column value and smaller than the end primary key column value are returned.
limit	The maximum number of returned rows. This parameter value must be greater than 0. An operation stops when the maximum number of rows are returned in the forward or backward direction, even if part of rows within the specified range are not returned. At this time, you can record the nextStartPrimaryKey value in the response for the next reading.
columnsToGet	The collection of columns you want to read. The columns can be primary key columns or attribute columns. If this parameter is not set, rows that contain all columns are returned. **Note** ◦ If you query a row of data, the system returns the data in all columns of the row. You can use the columnsToGet parameter to read data from specified columns. If col0 and col1 are added to columnsToGet, only the values of the col0 and col1 columns are returned. ◦ If the primary key set for a row is within the specified range, but the row does not contain the specified columns, the response does not contain the row of data. ◦ If columnsToGet and columnFilter are used at the same time, the system first obtains the columns specified by columnsToGet and then filters the returned columns.

Parameter	Description
maxVersions	The maximum number of versions you want to read. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data within a range or a version of data is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest.
timeRange	Reads data within a range of versions or a version of data. For more information, see TimeRange. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data within a range or a version of data is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ To query data within a range, you must set startTime and endTime. startTime specifies the start timestamp. endTime specifies the end timestamp. The time range is a left-closed and right-open interval, which is [startTime, endTime). ◦ To query data within a range of versions, set specificTime. specificTime specifies a specific timestamp. Only one of specificTime and [startTime, endTime) is required. Valid values: [0, Long.MAX_VALUE). Unit: milliseconds.
columnFilter	Filters read results on the server to return rows that meet the conditions in the filter. For more information, see Configure a filter. **Note** If columnsToGet and columnFilter are used at the same time, the system first obtains the columns specified by columnsToGet and then filters the returned columns.

Parameter	Description
nextStartPrimaryKey	The start primary key for the next reading. The value of nextStartPrimaryKey can be used to determine whether all data is read. ○ If the value of nextStartPrimaryKey is not empty in the response, the nextStartPrimaryKey value can be used as the start primary key for the next GetRange operation. ○ If the value of nextStartPrimaryKey is empty in the response, all data within the range is returned.

- Examples

  The following code provides an example on how to read data within a 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 data.next_start_primary_key is not empty, continue the read operation.
  if (data.next_start_primary_key) {
  }
  console.log('success:', data);
});
```

  For the detailed sample code, visit GetRange@GitHub.

5.8. Search Index

5.8.1. Create search indexes

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A data table whose timeToLive is set to -1 and maxVersions is set to 1 is created.

Parameters

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

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
fieldSchemas	The list of field schemas. You can configure the following parameters for each field schema: • fieldName: required. This parameter specifies the name of the field in the search index. The value is a column name. Type: String. A field in a search index can be a primary key column or an attribute column. • fieldType: required. This parameter specifies the type of the field. Use TableStore.FieldType.XXX to set the type. For more information, see Data types of column values. • index: optional. This parameter specifies whether to create an index for the column. Type: Boolean. By default, this parameter is set to true, and Tablestore creates an inverted index or spatial index for the column. If this parameter is set to false, Tablestore does not create indexes for the column. • analyzer: optional. This parameter specifies the type of analyzer to use. If fieldType is set to Text, you can set this parameter. If you do not specify this parameter, single-word tokenization is used. For more information about tokenization, see Tokenization. • enableSortAndAgg: optional. This parameter specifies whether to enable the sorting and aggregation features. Type: Boolean. Before you enable the sorting feature, you need to set enableSortAndAgg to true for a field. For more information about sorting, see Sorting and pagination. Notice • store: optional. This parameter specifies whether to store the value of the field in the search index. Type: Boolean. If you set the store parameter to true, you can read the value of the field directly from the search index without querying the data table. This improves query performance. • isAnArray: optional. This parameter specifies whether the value is an array. Type: Boolean. If you set this parameter to true, the column stores data as an array. Data written to the column must be a JSON array. Example: ["a","b","c"]. Nested values are considered as an array. If you set fieldType to Nested, skip the isAnArray parameter. • fieldSchemas: optional. This parameter specifies the list of field schemas for subfields. If the column is a Nested column, you must specify this parameter to configure the index types of subcolumns in the Nested column.

Parameter	Description
indexSetting	The settings of the search index, including routingFields. routingFields: optional. This parameter specifies custom routing fields. You can specify some primary key columns as routing fields. Tablestore 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.
indexSort	The presorting settings of the search index, including sorters. If indexSort is left empty, data is sorted by primary key. **Note** You can skip the presorting settings for search indexes that contain the Nested field type. sorters: required. This parameter specifies the presorting method for the search index. PrimaryKeySort and FieldSort are supported. For more information about sorting, see Sorting and pagination. • PrimaryKeySort: Data is sorted by primary key. You can configure the following parameter for PrimaryKeySort: order: the sort order. Data can be sorted in ascending or descending order. Default value: TableStore.SortOrder.SORT_ORDER_ASC. • FieldSort: Data is sorted by field value. You can configure the following parameters for FieldSort: You can presort field values only when a search index is created and the sorting and aggregation features are enabled for fields in the search index. ◦ fieldName: the name of the field to sort. ◦ order: the sort order. Data can be sorted in ascending or descending order. Default value: TableStore.SortOrder.SORT_ORDER_ASC. ◦ mode: the sorting method used when the field has multiple values.

Examples

```
/**
 *Create a search index that contains columns Col_Keyword, Col_Long, Col_Text, and Col_Nest
ed.
 *Set the type of data in Col_Keyword to Keyword, in Col_Long to Long, in Col_Text to Text,
and in Col_Nested to Nested.
 */
client.createSearchIndex({
    tableName: TABLE_NAME, // Set the name of the table.
    indexName: INDEX_NAME, // Set the name of the search index.
    schema: {
        fieldSchemas: [
            {
                fieldName: "Col_Keyword",
                fieldType: TableStore.FieldType.KEYWORD, // Set the name and type of the fi
eld.
                index: true, // Enable the indexing feature.
```

```
                enableSortAndAgg: true, // Enable the sorting and aggregation features.
                store: false,
                isAnArray: false
            },
            {
                fieldName: "Col_Long",
                fieldType: TableStore.FieldType.LONG,
                index: true,
                enableSortAndAgg: true,
                store: true,
                isAnArray: false
            },
            {
                fieldName: "Col_Text",
                fieldType: TableStore.FieldType.TEXT,
                index: true,
                enableSortAndAgg: false,
                store: true,
                isAnArray: false,
                analyzer: "single_word"
            },
            // {
            //     fieldName: "Col_Nested",
            //     fieldType: TableStore.FieldType.NESTED,
            //     index: false,
            //     enableSortAndAgg: false,
            //     store: false,
            //     fieldSchemas: [ // The subfield indexes set in Nested fields.
            //         {
            //             fieldName: "Sub_Col_KeyWord",
            //             fieldType: TableStore.FieldType.KEYWORD,
            //             index: true,
            //             enableSortAndAgg: true,
            //             store: false
            //         },
            //         {
            //             fieldName: "Sub_Col_Long",
            //             fieldType: TableStore.FieldType.LONG,
            //             index: true,
            //             enableSortAndAgg: true,
            //             store: false
            //         }
            //     ]
            // }
        ],
        indexSetting: { // The configuration options of the index.
            "routingFields": ["Pk_Keyword"], // You can only set the primary key column to
routingFields.
            "routingPartitionSize": null
        },
        indexSort: {// The search indexes that contain Nested fields do not support indexSo
rt.
            sorters: [
                // { // If indexSort is not set, data is sorted by primary key in ascending
order by default
```

```
order by default.
                //     primaryKeySort: {
                //         order: TableStore.SortOrder.SORT_ORDER_ASC
                //     }
                // },
                {
                    fieldSort: {
                        fieldName: "Col_Keyword",
                        order: TableStore.SortOrder.SORT_ORDER_DESC // Set the index sortin
g order.
                    }
                }
            ]
        }
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

5.8.2. List search indexes

You can call the ListSearchIndex operation to obtain indexes created in the current instance or associated with a table.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

```
client.listSearchIndex({
    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));
});
```

5.8.3. 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.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

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));
});
```

5.8.4. Delete search indexes

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.

- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the 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);
});
```

5.8.5. Term query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.

Parameter	Description
queryType	The query type. To use term query, set this parameter to TableStore.QueryType.TERM_QUERY.
fieldName	The name of the field used to match the query conditions.
term	The keyword used for the query to match the column values. The keyword is not tokenized but used as an exact word to match the column values. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that can match the keyword. For example, if the value of a TEXT field in a row is "tablestore is cool", the value can be tokenized into "tablestore", "is", and "cool". The row can meet query conditions when you set "tablestore", "is", or "cool" to the keyword to match the values of the TEXT field.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
columnToGet	Specifies whether to return all columns of each matched row. You can configure returnType and returnNames for this parameter. • If you set returnType to TableStore.ColumnReturnType.RETURN_SPECIFIED, you can use returnNames to specify the columns to return. • If you set returnType to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set returnType to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword exactly matches "hangzhou".
 */
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 n
umber of matched rows. A valve of 0 indicates that no data is returned.
        query: { // Set the query type to TableStore.QueryType.TERM_QUERY.
            queryType: TableStore.QueryType.TERM_QUERY,
            query: {
                fieldName: "Col_Keyword",
                term: "hangzhou"
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no attribute columns from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.6. Terms query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the table.

Parameter	Description
indexName	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
queryType	The query type. To use terms query, set this parameter to TableStore.QueryType.TERMS_QUERY.
fieldName	The name of the field used to match the query conditions.
terms	The keywords used in the query to match the column values. A row meets the query conditions when the value of the specified column exactly matches at least one keyword.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
columnToGet	Specifies whether to return all columns of each matched row. You can configure returnType and returnNames for this parameter. • If you set returnType to TableStore.ColumnReturnType.RETURN_SPECIFIED, you can configure returnNames to specify the columns to return. • If you set returnType to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set returnType to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * 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 the
se terms.
 */
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 n
umber of matched rows. A valve of 0 indicates that no data is returned.
        query: { // Set the query type to TableStore.QueryType.TERMS_QUERY.
            queryType: TableStore.QueryType.TERMS_QUERY,
            query: {
                fieldName: "Col_Keyword",
                terms: ["hangzhou", "shanghai"]
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no attribute columns from the table.
        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. Match all query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
query	The query type. To use match all query, set this parameter to TableStore.QueryType.MATCH_ALL_QUERY.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
columnToGet	Specifies whether to return all columns of each matched row. • If you set returnType to TableStore.ColumnReturnType.RETURN_SPECIFIED, you can configure returnNames to specify the columns to return. • If you set returnType to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set returnType to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * Use match all query 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 n
umber of matched rows. A valve of 0 indicates that no data is returned.
        query: {
            queryType: TableStore.QueryType.MATCH_ALL_QUERY
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no attribute columns from the table.
        returnType: TableStore.ColumnReturnType.RETURN_SPECIFIED,
        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));
});
```

5.8.8. Match query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Parameter	Description
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
queryType	The query type. To use match query, set this parameter to TableStore.QueryType.MATCH_QUERY.
fieldName	The name of the column used to match the query conditions. Match query applies to TEXT columns.
text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, you can specify a TEXT column for a query, and set the analyzer to perform single-word tokenization. If you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
operator	The logical operator. By default, OR is used as the logical operator, which indicates that a row matches the query conditions when one of the analyzed keywords is matched. If you set operator to AND, a row meets the query condition only when all tokenized keywords are matched.
minimumShouldMatch	The minimum number of matched tokenized keywords contained in a column value. A row is returned only when the value of the queried column in the row contains at least the minimum number of matched tokenized keywords. **Note** minimumShouldMatch must be used together with the OR logical operator.

Parameter	Description
getTotalCount	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
columnToGet	Specifies whether to return all columns of each matched rows. You can configure returnType and returnNames for this parameter. • If you set returnType to TableStore.ColumnReturnType.RETURN_SPECIFIED, you can use returnNames to specify the columns to return. • If you set returnType to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set returnType to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword matches "hangzhou". Tablestore
returns the number of rows that match the phrase and parts of the matched rows in this quer
y.
 */
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 n
umber of matched rows. The valve of 0 indicates that no data is returned.
        query: { // Set the query type to MatchQuery.
            queryType: TableStore.QueryType.MATCH_QUERY,
            query: {
                fieldName: "Col_Keyword", // Set the column used to match the query conditi
ons
                text: "hangzhou" // Set the keyword used for the query to match the column
values.
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no data from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.9. Phrase query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
queryType	The query type. To use phrase query, set this parameter to TableStore.QueryType.MATCH_PHRASE_QUERY.
fieldName	The name of the column used to match the query conditions. Match query applies to TEXT columns.
text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore" and "this is a table". Query results such as "this table is ..." or "is this a table" cannot be obtained.
getTotalCount	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
columnToGet	Specifies whether to return all columns of each matched rows. • If you set returnType to TableStore.ColumnReturnType.RETURN_SPECIFIED, you can use returnNames to specify the columns to return. • If you set returnType to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set returnType to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Text matches the whole phase "hangzhou
shanghai" in order.
 * Tablestore returns the number of rows that match the phrase and parts of the matched row
s 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 n
umber of matched rows. The valve of 0 indicates that no data is returned.
        query: {// Set the query type to TableStore.QueryType.MATCH_PHRASE_QUERY.
            queryType: TableStore.QueryType.MATCH_PHRASE_QUERY,
            query: {
                fieldName: "Col_Text", // Set the column used to match the query conditions
.
                text: "hangzhou shanghai" // Set the keyword used for the query to match th
e column values.
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no data from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.10. Prefix query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
queryType	The query type. To use prefix query, set this parameter to TableStore.QueryType.PREFIX_QUERY.
fieldName	The name of the field used to match the query conditions.
prefix	The value of the prefix. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that contains the specified prefix.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
columnToGet	Specifies whether to return all columns of each matched row. You can configure returnType and returnNames for this parameter. • If you set returnType to TableStore.ColumnReturnType.RETURN_SPECIFIED, you can configure returnNames to specify the columns to return. • If you set returnType to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set returnType to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * Search the table for rows where the value of Col_Keyword contains the prefix that exactl
y matches "hang", such as "hangzhou".
 */
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 n
umber of matched rows. A valve of 0 indicates that no data is returned.
        query: { // Set the query type to TableStore.QueryType.PREFIX_QUERY.
            queryType: TableStore.QueryType.PREFIX_QUERY,
            query: {
                fieldName: "Col_Keyword",
                prefix: "hang" // Set the prefix to "hang" to match terms such as "hangzhou
" or "hangzhoushi".
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no attribute columns from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.11. Range query

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Parameter	Description
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
queryType	The query type. To use prefix query, set this parameter to TableStore.QueryType.RANGE_QUERY.
fieldName	The name of the field used to match the query conditions.
rangeFrom	The value from which the query starts.
rangeTo	The value with which the query ends.
includeLower	Specifies whether to include the value of rangeFrom in the response. The value type is Boolean.
includeUpper	Specifies whether to include the value of rangeTo in the response. The value type is Boolean.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
columnToGet	Specifies whether to return all columns of each matched row. You can configure returnType and returnNames for this parameter. • If you set returnType to TableStore.ColumnReturnType.RETURN_SPECIFIED, you can configure returnNames to specify the columns to return. • If you set returnType to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set returnType to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * Search the table for rows where the values of Col_Long ranges from 1 to 10, and the rang
e includes 1 but excludes 10. Tablestore sorts these rows by the value of Col_Long in desce
nding 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 n
umber of matched rows. A valve of 0 indicates that no data is returned.
        query: { // Set the query type to TableStore.QueryType.RANGE_QUERY.
            queryType: TableStore.QueryType.RANGE_QUERY,
            query: {
                fieldName: "Col_Long",
                rangeFrom: 1,
                includeLower: true, // Include the value of rangeFrom in the range.
                rangeTo: 10,
                includeUpper: false // Exclude the value of rangeTo from the range.
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no attribute columns from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.12. Wildcard query

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

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the data table.
indexName	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that you want the current query to return.
queryType	The query type. Set this parameter to TableStore.QueryType.WILDCARD_QUERY.
fieldName	The name of the column.
value	The string that contains wildcards. The string cannot exceed 32 characters in length.
getTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, the value of this parameter is false, which specifies that the total number of rows that match the query conditions is not returned. If you set the getTotalCount parameter to true, the query performance is compromised.
columnToGet	Specifies whether to return all columns of each matched row. • If you set the returnType parameter to TableStore.ColumnReturnType.RETURN_SPECIFIED, you must specify the columns that you want to return. • If you set the returnType parameter to TableStore.ColumnReturnType.RETURN_ALL, all columns are returned. • If you set the returnType parameter to TableStore.ColumnReturnType.RETURN_NONE, only the primary key columns are returned.

Examples

```
/**
 * Search the table for rows in which the value of the Col_Keyword column matches "table*e"
.
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, // If you want to obtain the number of matched rows without returning sp
ecific data, you can set limit to 0. A value of 0 specifies that no data is returned.
        query: { // Set the query type to TableStore.QueryType.WILDCARD_QUERY.
            queryType: TableStore.QueryType.WILDCARD_QUERY,
            query: {
                fieldName: "Col_Keyword",
                value: "table*e" // Specify a string that contains one or more wildcard cha
racters in wildcardQuery.
            }
        },
        getTotalCount: true // The value of TotalCount specifies the total number of rows t
hat match the query conditions. By default, the value of getTotalCount is false, which spec
ifies that the total number of rows that match the query conditions is not returned. In thi
s example, the getTotalCount parameter is set to true.
    },
    columnToGet: { // You can use RETURN_SPECIFIED to return the columns that you want to r
eturn, RETURN_ALL to return all columns, or RETURN_NONE to return only the primary key colu
mns.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.13. Geo query

Geo query includes the following query types: geo-distance query (GeoDistanceQuery), geo-bounding box query (GeoBoundingBoxQuery), and geo-polygon query (GeoPolygonQuery).

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Geo-distance query

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

- Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
query	The query statement for the search index. Sets the query type to TableStore.QueryType.GEO_DISTANCE_QUERY.
fieldName	The column name. Type: Geopoint.
centerPoint	The coordinate pair of the central point. The coordinate pair consists of latitude and longitude values. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of the longitude: [-180,180]. Example: "35.8,-45.91".
distance	The radius of the circle centered on the specified location. Type: Double. Unit: meters.

- Examples

  The following code provides an example on how to search the table for rows where the value of Col_GeoPoint falls within a specified distance from a specified central point:

```
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 TableStore.QueryType.GEO_DISTANCE_QUERY.
            queryType: TableStore.QueryType.GEO_DISTANCE_QUERY,
            query: {
                fieldName: "Col_GeoPoint",
                centerPoint: "1,1", // Specify the coordinate pair for a central point.
                distance: 10000 // You can set the distance from the central point to a v
alue greater than or equal to 10,000. Unit: meters.
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that
match the query conditions. By default, TotalCount is set to false, which indicates that
the total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETUR
N_ALL to return all columns, or RETURN_NONE to return no data from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

Geo-bounding box query

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

- Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
query	The query statement for the search index. Sets the query type to TableStore.QueryType.GEO_BOUNDING_BOX_QUERY.
fieldName	The column name. Type: Geopoint.
topLeft	The coordinate pair of the upper-left corner of the rectangular geographic area.

Parameter	Description
bottomRight	The coordinate pair of the lower-right corner of the rectangular geographic area. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of the longitude: [-180,180]. Example: "35.8,-45.91".

- Examples

  The data type of Col_GeoPoint is GeoPoint. The coordinate pair of the upper-left corner is "10,0". The coordinate pair of the lower-right corner is "0,10". The following code provides an example on how to obtain the rows where the value of Col_GeoPoint falls within the rectangular geographic area:

```
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 TableStore.QueryType.GEO_BOUNDING_BOX_QUERY.
            queryType: TableStore.QueryType.GEO_BOUNDING_BOX_QUERY,
            query: {
                fieldName: "Col_GeoPoint", // Set the column name.
                topLeft: "10,0", // Specify the coordinate pair for the upper-left corner of the rectangular geographic area.
                bottomRight: "0,10" // Specify coordinate pair for the lower-right corner of the rectangular geographic area.
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that match the query conditions. By default, TotalCount is set to false, which indicates that the total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_ALL to return all columns, or RETURN_NONE to return no data from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

Geo-polygon query

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

- Parameters

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

- Examples

  The following code provides an example on how to query the table for rows where the value of Col_GeoPoint falls within a specified geographic polygon area:

```
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 TableStore.QueryType.GEO_POLYGON_QUERY.
            queryType: TableStore.QueryType.GEO_POLYGON_QUERY,
            query: {
                fieldName: "Col_GeoPoint",
                points: ["0,0","5,5","5,0"] //  Specify coordinate pairs for vertices of a geographic polygon area.
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that match the query conditions. By default, TotalCount is set to false, which indicates that the total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_ALL to return all columns, or RETURN_NONE to return no data from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.14. Boolean query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the data table.
indexName	The name of the search index.
mustQueries	The list of subqueries that the query results must match. This parameter is equivalent to the AND operator.
mustNotQueries	The list of subqueries that the query results must not match. This parameter is equivalent to the NOT operator.
filterQueries	The list of subqueries. Only rows that match all subfilters are returned. filter is similar to query except that filter does not calculate the relevance score based on the number of subfilters that the row matches.
shouldQueries	The list of subqueries that the query results can or cannot match. This parameter is equivalent to the OR operator. Only rows that meet the minimum number of subquery conditions specified by shouldQueries are returned. A higher overall relevance score indicates that more subquery conditions specified by shouldQueries are met.
minimumShouldMatch	The minimum number of subquery conditions specified by shouldQueries that the rows must meet. If no other subquery conditions except the subquery conditions that are specified by shouldQueries are specified, the default value of the minimumShouldMatch parameter is 1. If other subquery conditions, such as subquery conditions specified by mustQueries, mustNotQueries, and filterQueries are specified, the default value of the minimumShouldMatch parameter is 0.

Examples

The following code provides an example on how to construct a Boolean query to query the rows based on a combination of subqueries:

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

```
var client = require('../client');
var TableStore = require('../../index.js');
var Long = TableStore.Long;
/**
 * Perform a Boolean query to implement the following operations: (col2 < 4 or col 3 < 5) o
r (col2 = 4 and (col3 = 5 or col3 = 6)). Logic:
 * boolQuery1 = rangeQuery(col2<4) or rangeQuery(col3<5)
 * boolQuery2 = termQuery(col3=5) or (col3=6)
 * boolQuery3 = termQuery(col2=4) and boolquery2
 * boolQuery4 = boolQuery1 or boolQuery3
 */
client.search({
    tableName: "sampleTable",
    indexName: "sampleSearchIndex",
    searchQuery: {
        offset: 0, // Query the offset value.
        limit: 10, // To query the number of only matched rows without any detailed data, y
ou can set limit to 0. In this case, Tablestore returns the number of matched rows without
data from the table.
        getTotalCount: false, // The value of TotalCount indicates the total number of rows
that meet the query conditions. By default, getTotalCount is false, which indicates that th
e total number of matched rows is not returned. Retain the default value in this example.
        query: { // Construct boolQuery4. Specify the query condition to meet at least one
of boolQuery1 and boolQuery3.
            queryType: TableStore.QueryType.BOOL_QUERY,
            query: {
                shouldQueries: [ // Specify mustQueries, shouldQueries, or mustNotQueries.
                    { // Construct boolQuery1. Specify the query condition to meet at least
one of Query Condition 1 and Query Condition 2.
                        queryType: TableStore.QueryType.BOOL_QUERY,
                        query: {
                            // Specify shouldQueries to query the rows that contain the col
2 column whose value is smaller than 4 or contain the col3 column whose value is smaller th
an 5.
                            shouldQueries:[
                                {
                                    // Query Condition 1: Perform a range query to query th
e rows that contain the col2 column whose value is smaller than 4.
                                    queryType: TableStore.QueryType.RANGE_QUERY,
                                    query:{
                                        fieldName: "col2",
                                        rangeTo: 4
                                    }
                                },
                                {
                                    // Query Condition 2: Perform a range query to query th
e rows that contain the col3 column whose value is smaller than 5.
queryType: TableStore.QueryType.RANGE_QUERY,
                                    query:{
                                        fieldName: "col3",
                                        rangeTo: 5
                                    }
                                }
                            ],
                            minimumShouldMatch:1
```

```
                        }
                    },
                    { // Construct boolQuery3. Specify the query condition to meet the cond
itions of Query Condition 3 and boolQuery2.
                        queryType: TableStore.QueryType.BOOL_QUERY,
                        query: {
                            mustQueries: [
                                // Specify mustQueries to query the rows that contain the c
ol2 column whose value is equal to 4 and contain the col3 column whose value is equal to 5
or 6.
                                {
                                    // Query Condition 3: Perform a term query to query the
rows that contain the col2 column whose value is equal to 4.
                                    queryType:TableStore.QueryType.TERM_QUERY,
                                    query: {
                                        fieldName : "col2",
                                        term: 4
                                    }
                                },
                                { // Construct boolQuery2: Specify the query condition to m
eet at least one of Query Condition 4 and Query Condition 5.
                                    queryType: TableStore.QueryType.BOOL_QUERY,
                                    query: {
                      TERM                  // Specify shouldQueries to query the rows that
contain the col3 column whose value is equal to 5 or 6.
                                        shouldQueries:[
                                            {
                                                // Query Condition 4: Perform a term query
to query the rows that contain the col3 column whose value is equal to 5.
                                                queryType: TableStore.QueryType.TERM_QUERY,
                                                query:{
                                                    fieldName:"col3",
                                                    term: 5
                                                }
                                            },
                                            {
                                                // Query Condition 5: Perform a term query
to query the rows that contain the col3 column whose value is equal to 6.
queryType: TableStore.QueryType.TERM_QUERY,
                                                query:{
                                                    fieldName:"col3",
                                                    term: 6
                                                }
                                            }
                                        ],
                                        minimumShouldMatch:1
                                    }
                                }
                            ]
                        }
                    }
                ],
                minimumShouldMatch: 1 // Specify the minimum number of conditions that must
be met. This parameter is valid when subquery conditions are specified only by shouldQuerie
```

```
s.
            }
        },
    },
    columnToGet: { // Set RETURN_SPECIFIED to return the specified columns, RETURN_ALL to r
eturn all columns, or RETURN_NONE to return no attribute columns.
        returnType: TableStore.ColumnReturnType.RETURN_SPECIFIED,
        returnNames: ["col2", "col3", "col4"]
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.15. Nested query

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
tableName	The name of the table.
indexName	The name of the search index.
path	The path of the NESTED column. The path is similar to the tree structure. For example, news.title indicates the title subcolumn in the news column of the NESTED type.
query	The query on the subcolumn in the NESTED column. The query can be of any query type.
scoreMode	The value on which a score is calculated when multiple values exist for a column.

Examples

The following code provides an example on how to search "Hangzhou" of the Col_Nested.Sub_Col_Keyword column for data:

```
/**
 * 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 n
umber of rows. The valve of 0 indicates that no data is returned.
        query: { // Set the query type to TableStore.QueryType.NESTED_QUERY.
            queryType: TableStore.QueryType.NESTED_QUERY,
            query: {
                path: "Col_Nested",
                query: {
                    queryType: TableStore.QueryType.TERM_QUERY,
                    query: {
                        fieldName: "Col_Nested.Sub_Col_Keyword",
                        term: "Hangzhou"
                    }
                },
            }
        },
        getTotalCount: true // The value of TotalCount indicates the number of rows that ma
tch the query conditions. By default, TotalCount is set to false, which indicates that the
total number of matched rows is not returned. Set TotalCount to true in this example.
    },
    columnToGet: { // You can set RETURN_SPECIFIED to return the specified columns, RETURN_
ALL to return all columns, or RETURN_NONE to return no data from the table.
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

5.8.16. Sorting and pagination

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

Index presorting

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

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

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

Specify a sorting method

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

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

- ScoreSort

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

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

  ```
  sort: {
      sorters: [
          {
              scoreSort: {
                  order: TableStore.SortOrder.SORT_ORDER_ASC
              }
          }
      ]
  }
  ```

- PrimaryKeySort

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

  ```
  sort: {
      sorters: [
          {
              primaryKeySort: {
                  order: TableStore.SortOrder.SORT_ORDER_DESC // Sort the query result in t
  he descending order.
                  //order: TableStore.SortOrder.SORT_ORDER_ASC // Sort the query result in
  the ascending order.
              }
          }
      ]
  }
  ```

- FieldSort

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

```
sort: {
    sorters: [
        {
            fieldSort: {
                fieldName: "Col_Keyword",
                order: TableStore.SortOrder.SORT_ORDER_DESC
            }
        }
    ]
}
```

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

```
sort: {
    sorters: [
        {
            fieldSort: {
                fieldName: "Col_Keyword",
                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 results by geographical location.

```
sort: {
    sorters: [
        {
            geoDistanceSort: {
                fieldName: "Col_Geo_Point",
                points: ["0,0"],// Set the central point.
                order: TableStore.SortOrder.SORT_ORDER_ASC // The query result is returne
d in ascending order of the distances between geographical locations and the central poin
t.
            }
        }
    ]
}
```

For the detailed sample code, see Search.

Specify a pagination method

You can use limit and offset or use tokens to split returned query results into pages.

- Use the limit and offset parameters

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

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

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

```
/**
 * Set offset to 90 and limit to 10 to scan data from line 90 to line 99.
 */
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 tha
t meet the query condition is returned. If TotalCount is set to false, the total number o
f rows that meet the query condition is not returned.
    },
    columnToGet: { // Set RETURN_SPECIFIED to return the specified 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));
});
```

- Use a token

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

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

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

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

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

```
/**
 * The following code provides an example on how to use tokens to paginate results in synchronous and asynchronous modes.
 */
var params = {
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10,
        token: null,// Obtain the nextToken as the start point to scan the next page. The data type is byte stream.
        query: {
            queryType: TableStore.QueryType.MATCH_ALL_QUERY
        },
        getTotalCount: true
    },
    columnToGet: {
        returnType: TableStore.ColumnReturnType.RETURN_SPECIFIED,
        returnNames: ["pic_tag", "pic_description", "time_stemp", "pos"]
    }
};
/**
 * The following code provides an example on how to use tokens to paginate results in synchronous mode.
 */
(async () => { // Code in synchronous mode.
  try {
    var data = await client.search(params);
    console.log('success:', JSON.stringify(data, null, 2));
    while (data.nextToken && data.nextToken.length) { // If nextToken is returned, Tablestore continues to read data.
      // Store the tokens persistently.
      // Convert a token (nextToken) as binary data in the Buffer object to a Base64 string and store the string persistently.
      // You can convert the persisted Base64 string to binary data by creating a Buffer object to use nextToken as a parameter.
      var nextToken = data.nextToken.toString("base64");
      var token =  Buffer.from(nextToken, "base64");
      params.searchQuery.token = token;// Update the token value to return more results.
      data = await client.search(params);
      console.log('token success:', JSON.stringify(data, null, 2));
```

```
    }
  } catch (error) {
      console.log(error);
  }
})()
/**
 * The following code provides an example on how to use tokens to paginate results in asy
nchronous mode.
 */
client.search(params, function (err, data) {
    console.log('success:', JSON.stringify(data, null, 2));
    if (data.nextToken && data.nextToken.length) {
        // Store the tokens persistently.
        // Convert a token (nextToken) as binary data in the Buffer object to a Base64 st
ring and store the string persistently.
        // You can convert the persisted Base64 string to binary data by creating a Buffe
r object to use nextToken as a parameter.
        var nextToken = data.nextToken.toString("base64");
        var token =  Buffer.from(nextToken, "base64");
        params.searchQuery.token = token;// Update the token value to return more results
.
        client.search(params, function (err, data) {
            console.log('token success:', JSON.stringify(data, null, 2));
        });
    }
});
```

5.9. Secondary Index

5.9.1. Global secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created. The value of timeToLive is -1. The value of maxVersions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table for an existing data table.

> **Note** You can call the CreateTable operation to create one or more index tables when you create a data table. For more information, see Create data tables.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
indexMeta	The schema information of the index table. The schema information contains the following items: ○ name: the name of the index table. ○ primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ○ definedColumn: the attribute columns of the index table. The attribute columns are a combination of predefined columns of the data table. ○ includeBaseData: specifies whether to include the existing data from the data table in the index table. If includeBaseData parameter is set to true, the index table contains the existing data. If the value is set to false, the index table does not contain the existing data. ○ indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to index and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. ○ indexUpdateMode: the update mode of the index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ■ If indexUpdateMode is not specified or is set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the global secondary index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ■ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the index update mode to IUM_SYNC_INDEX.

Parameter	Description

- Examples

```
var client = require('./client');
var TableStore = require('../index.js');
client.createIndex({
    mainTableName: "sdkGlobalIndexTest",
    indexMeta: {
        name: "sdkGlobalIndex",
        primaryKey: ["col1"],
        definedColumn: ["col2"],
        includeBaseData: false,
        indexUpdateMode: TableStore.IndexUpdateMode.IUM_ASYNC_INDEX,// By default, the in
dex update mode is set to IUM_ASYNC_INDEX.
        indexType: TableStore.IndexType.IT_GLOBAL_INDEX,// By default, the index type is
set to IT_GLOBAL_INDEX.
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  - Parameters

    When you call the GetRow operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Examples

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "index1",
  primaryKey: [ {'pk2': Long.fromNumber(2)}, {'pk1': Long.fromNumber(1)}]
};
client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

- Read data within a specified range from an index table

  For more information, see Multi-row operations.

  - Parameters

    When you call the GetRange operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

  - Examples

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sdkIndex1", // In the example, pk2 is the first primary key column of the
index table. pk1 is the second primary key column of the index table.
  direction: TableStore.Direction.FORWARD,
  maxVersions: 10,
  inclusiveStartPrimaryKey: [{ "pk2": TableStore.INF_MIN }, { "pk1": TableStore.INF_MIN
}],
  exclusiveEndPrimaryKey: [{ "pk2": TableStore.INF_MAX }, { "pk1": TableStore.INF_MAX }
],
  limit: 2
};
var resultRows = []
var getRange = function () {
  client.getRange(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    resultRows = resultRows.concat(data.rows)
    // If data.next_start_primary_key is not empty, the system continues to read data.
    if (data.nextStartPrimaryKey) {
      params.inclusiveStartPrimaryKey = [
        { "pk2": data.nextStartPrimaryKey[0].value },
        { "pk1": data.nextStartPrimaryKey[1].value }
      ];
      getRange()
    } else {
      console.log(JSON.stringify(resultRows));
    }
  });
}
getRange()
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
indexName	The name of the index table.

- Examples

```
var client = require('./client');
client.dropIndex({
  mainTableName: "sdkGlobalTest",
  indexName: "sdkIndex1"
}, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

5.9.2. Local secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- Client is initialized. For more information, see Initialization.
- A data table is created. The value of timeToLive is -1. The value of maxVersions is 1.
- Predefined columns are configured for the data table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table for an existing data table.

> **Note** You can call the CreateTable operation to create one or more index tables when you create a data table. For more information, see Create data tables.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
	The schema information of the index table. The schema information contains the following items: ○ name: the name of the index table. ○ primaryKey: the indexed columns of the index table. The indexed columns are a combination of primary key columns and predefined columns of the data table. If you use the local secondary index feature, the first primary key column of an index table must be the same as the first primary key column of the corresponding data table. ○ definedColumn: the attribute columns of the index table. The attribute columns are a combination of predefined columns of the data table.

Parameter	Description
indexMeta	○ includeBaseData: specifies whether to include the existing data from the data table in the index table. If includeBaseData parameter is set to true, the index table contains the existing data. If the value is set to false, the index table does not contain the existing data. ○ indexType: the type of the index. Valid values: IT_GLOBAL_INDEX and IT_LOCAL_INDEX. ■ If indexType is not specified or is set to IT_GLOBAL_INDEX, the global secondary index feature is used. If you use the global secondary index feature, Tablestore automatically synchronizes the columns to index and data in primary key columns from a data table to an index table in asynchronous mode. The synchronization latency is within a few milliseconds. ■ If indexType is set to IT_LOCAL_INDEX, the local secondary index feature is used. If you use the local secondary index feature, Tablestore automatically synchronizes data from the indexed columns and the primary key columns of a data table to the columns of an index table in synchronous mode. After the data is written to the data table, you can query the data from the index table. ○ indexUpdateMode: the update mode of the index. Valid values: IUM_ASYNC_INDEX and IUM_SYNC_INDEX. ■ If indexUpdateMode is not specified or is set to IUM_ASYNC_INDEX, the asynchronous mode is used to update the global secondary index. If you use the global secondary index feature, you must set the index update mode to IUM_ASYNC_INDEX. ■ If you set indexUpdateMode to IUM_SYNC_INDEX, the synchronous update mode is used. If you use the local secondary index feature, you must set the index update mode to IUM_SYNC_INDEX.

- Examples

```
var client = require('./client');
var TableStore = require('../index.js');
client.createIndex({
    mainTableName: "sdkLocalIndexTest",
    indexMeta: {
        name: "sdkLocalIndex",
        primaryKey: ["pk1","col1"],// Add primary key columns to the index table. The fir
st primary key column of the index table must be the same as the first primary key column
of the data table.
        definedColumn: ["col2"],
        includeBaseData: false,
        indexUpdateMode: TableStore.IndexUpdateMode.IUM_SYNC_INDEX,// Set the index updat
e mode to IUM_SYNC_INDEX, which specifies the synchronous update mode. If the index type
is set to IT_LOCAL_INDEX, the index update mode must be set to IUM_SYNC_INDEX.
        indexType: TableStore.IndexType.IT_LOCAL_INDEX,// Set the index type to IT_LOCAL_
INDEX, which specifies a local secondary index.
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

Read data from an index table

You can read a row of data or read data within a specified range from the index table. If the index table contains the attribute columns to return, you can query the data from the index table. If the index table does not contain the columns to return, you must query the required data from the data table.

- Read a row of data from an index table

  For more information, see Single-row operations.

  - Parameters

    When you call the GetRow operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the primary key columns of a row, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

- Examples

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sdkLocalIndex",
  primaryKey: [ {'pk1': Long.fromNumber(1)},{'col1': Long.fromNumber(2)}, {'pk2': Long.
fromNumber(2)}]
};
client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

- Read data within a specified range from an index table

  For more information, see Multi-row operations.

  - Parameters

    When you call the GetRange operation to read data from an index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore automatically adds the primary key columns that are not specified as indexed columns to an index table. Therefore, when you specify the start primary key and end primary key, you must specify the indexed columns based on which you create the index table and the primary key columns of the data table.

  - Examples

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sdkLocalIndex",
  direction: TableStore.Direction.FORWARD,
  maxVersions: 10,
  inclusiveStartPrimaryKey: [{ "pk1": TableStore.INF_MIN }, { "col1": TableStore.INF_MI
N }, { "pk2": TableStore.INF_MIN }],
  exclusiveEndPrimaryKey: [{ "pk1": TableStore.INF_MIN }, { "col1": TableStore.INF_MIN
}, { "pk2": TableStore.INF_MAX }],
  limit: 2
};
var resultRows = []
var getRange = function () {
  client.getRange(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    resultRows = resultRows.concat(data.rows)
    // If data.next_start_primary_key is not empty, the system continues to read data.
    if (data.nextStartPrimaryKey) {
      params.inclusiveStartPrimaryKey = [
        { "pk1": data.nextStartPrimaryKey[0].value },
        { "col1": data.nextStartPrimaryKey[1].value }
        { "pk2": data.nextStartPrimaryKey[2].value }
      ];
      getRange()
    } else {
      console.log(JSON.stringify(resultRows));
    }
  });
}
getRange()
```

Delete an index table by calling the DeleteIndex operation

You can call the DeleteIndex operation to delete the specified index table from the corresponding data table.

- Parameters

Parameter	Description
mainTableName	The name of the data table.
indexName	The name of the index table.

- Examples

```
var client = require('./client');
client.dropIndex({
  mainTableName: "sdkGlobalTest",
  indexName: "sdkIndex1"
}, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

5.10. Error handling

This topic describes the error handling methods and retry policies of Tablestore SDK for Node.js.

Methods

Tablestore SDK for Node.js handles errors as exceptions. If the operation succeeds, the called operation does not return an exception. If the operation fails, an exception is returned.

> **Note** Batch operations such as BatchGetRow and BatchWriteRow are successful only when the system verifies that no exception is returned and that the status of each row is successful.

Exceptions

All errors are handled by Tablestore SDK for Node.js in a unified manner, and are returned as the value of the err parameter by using the callback method. Before you obtain the returned data, you must check whether the err parameter has a value. If an error occurs at the server side of Tablestore, the request ID is returned. The request ID is the UUID that identifies a request. If a problem persists, record this request ID and submit a ticket.

Retry

Tablestore SDK for Node.js automatically retries operations in which an error occurs. In the default retry policy, the maximum retry attempts is 20, and the maximum retry interval is 3,000 ms. For more information about the retry policies for throttling errors and internal server errors related to read operations, see tablestore/lib/retry.js.

6..NET SDK

6.1. Preface

This topic describes how to use Tablestore SDK for .NET.

Prerequisites

- Tablestore is activated. For more information, see Activate Tablestore.
- An AccessKey pair is created. For more information, see Create an AccessKey pair for a RAM user.

Download Tablestore SDK for .NET

- Download the SDK installation package from NuGet. For more information about the download path, visit .NET SDK 4.1.4.
- Download the source code from GitHub. For more information about the download path, visit GitHub.

For more information about the historical versions of Tablestore SDK for .NET, see Tablestore SDK for .NET.

Compatibility

For SDKs V3.x.x: compatible.

For SDKs V2.x.x:

- Part of operations are incompatible. The Condition.IGNORE, Condition.EXPECT_EXIST, and Condition.EXPECT_NOT_EXIST parameters are deleted.
- The DLL file name is changed from Aliyun.dll to Aliyun.TableStore.dll.

Version

Latest version: 4.1.4.

6.2. Installation

This topic describes how to install Tablestore SDK for .NET.

Version dependencies

Windows

- .NET 4.x is required.
- Visual Studio 2010 or later is required.

Install Tablestore SDK for .NET in Windows

- Install by using NuGet
  i. After you install NuGet, create a project or open an existing project in Visual Studio. Then, choose **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.

> **Note** If Visual Studio does not have NuGet installed, download and install NuGet. For more information about the download path, visit NuGet.

ii. Enter aliyun.tablestore in the search box and click Search. Find Aliyun.TableStore.SDK in the search results.

iii. Select the latest version, and click **Install**.

After the installation is complete. Tablestore SDK for .NET is added to the project for application.

- Install by importing a project

i. Download the source code from GitHub. For more information about the download path, visit GitHub.

> **Note** If Git is not installed, download and install Git. For more information about the download path, visit git.

ii. In Visual Studio, right-click and select **Solution** from the menu that appears, choose **Add > Existing projects**.

iii. In the dialog box that appears, select the *aliyun-tablestore-sdk.csproj* file. Click **Open**.

iv. Right-click and select **Your project**. Choose **References > Add Reference** from the shortcut menu. In the dialog box that appears, click the **Projects** tab, and select the aliyun-tablestore-sdk project.

v. Click **OK**.

6.3. Initialization

OTSClient is the client for Tablestore. OTSClient provides a series of methods for you to manage tables and perform read and write operations on one or more rows.

Obtain an endpoint

An endpoint is a domain name that is used to access a Tablestore instance in a region. To query an endpoint, perform the following steps:

1. Log on to the Tablestore console.
2. Click the name of the instance to go to the **Instance Details** tab.

   On the Instance Details tab, you can view the endpoints of the instance in the Instance Access URL section.

> **Note** For more information about endpoints, see Endpoint.

Configure an AccessKey pair

To access Tablestore, you must have an AccessKey pair that consists of an AccessKey ID and AccessKey secret to verify your identity. The following types of AccessKey pairs are supported:

- The AccessKey pair of an Alibaba Cloud account. To obtain the AccessKey pair of an Alibaba Cloud account, take the following steps:

i. On the Alibaba Cloud official website,Create Your Alibaba Cloud Account.

ii. Create an AccessKey ID and an AccessKey secret. For more information, see Obtain an AccessKey pair.

- The AccessKey pair of a RAM user who is granted with permissions to access Tablestore. To obtain

the AccessKey pair of a RAM user, take the following steps:

i. Use your Alibaba Cloud account to log on to RAM. Create a RAM user or use an existing RAM user.

ii. Use your Alibaba Cloud account to authorize the RAM user to access Tablestore.

iii. After the RAM user is authorized, you can use the AccessKey pair of the RAM user to access Tablestore.

- Temporary access credentials obtained from STS. To obtain temporary access credentials from STS, take the following steps:

  i. The application server uses RAM or STS to obtain access credentials that consist of a temporary AccessKey ID, an AccessKey secret, and a token. After the access credentials are obtained, the application server sends them to you.

  ii. You can use the received access credentials to access Tablestore.

Initializa an OTSClient instance

After you obtain the AccessKey ID and AccessKey secret, you can perform the following operations to initialize a Tablestore client instance:

- Operations

```
    /// <summary>
    /// The function used to construct OTSClient.
    /// </summary>
    /// <param name="endPoint"> The endpoint of the Tablestore instance, which must start with https://. Example: https://instance.cn-hangzhou.ots.aliyun.com:80 </param>
    /// <param name="accessKeyID"> The AccessKey ID that is used to access the Tablestore instance. You can apply for an AccessKey ID on the Alibaba Cloud official website. </param>
    /// <param name="accessKeyID"> The AccessKey secret that is used to access the Tablestore instance. You can apply for an AccessKey secret on the Alibaba Cloud official website. </param>
    /// <param name="instanceName"> The name of the Tablestore instance. You can customize the instance name in the Tablestore console. </param>
public OTSClient(string endPoint, string accessKeyID, string accessKeySecret, string instanceName);
    /// <summary>
    /// Create an OTSClient instance by configuring the instance of the OTSClientConfig class on the client.
    /// </summary>
    /// <param name="config"> The instance for client configurations.</param>
    public OTSClient(OTSClientConfig config);
```

- Examples

> **Note**
>
> - You can set the ConnectionLimit parameter in the OTSClientConfig instance. If you do not specify the value of the ConnectionLimit parameter, the default value 300 is used.
> - You can set OTSDebugLogHandler and OTSErrorLogHandler in OTSClientConfig to specify whether to enable logging.
> - You can specify RetryPolicy in OTSClientConfig to control and customize the retry logic. You can also use the default retry policy.

```
    // Create an OTSClientConfig object.
    var config = new OTSClientConfig(Endpoint, AccessKeyId, AccessKeySecret, InstanceName
);
    // Disable logging. By default, logging is enabled.
    config.OTSDebugLogHandler = null;
    config.OTSErrorLogHandler = null;
    // Create an OTSClient object by using OTSClientConfig.
    var otsClient = new OTSClient(config);
    // Use OTSClient to insert or query data.
```

Multithreading

- Multithreading is supported.
- We recommend that you use the same OTSClient object to perform multithreading tasks.

6.4. Table

6.4.1. Create tables

This topic describes how to create a data table by calling the CreateTable operation. When you call the CreateTable operation, you must specify the schema information and configuration information of the data table when you call the CreateTable operation. You can set reserved read throughput and reserved write throughput for data tables in high-performance instances. You can create one or more index tables when you create a data table.

> **Note**
>
> - After a data table is created, it takes several seconds to load the data table. During this period, read and write operations performed on the table may fail. Perform operations on the data table after the data table is loaded.
> - You must specify the primary key when you create a data table. A primary key can contain one to four primary key columns. Each primary key column has a name and a data type.

Prerequisites

- An instance is created in the console. For more information, see Create instances.
- OTSClient is initialized. For more information, see Initialization.

Operations

```
/// <summary>
 /// Create a table based on table information including the table name, primary key schema
, and reserved read and write throughput.
 /// </summary>
 /// <param name="request">Request parameter</param>
 /// <returns>Information returned by CreateTable. The result is null.
 /// </returns>
        public CreateTableResponse CreateTable(CreateTableRequest request);
        /// <summary>
        /// Asynchronous mode of CreateTable.
        /// </summary>
        public Task<CreateTableResponse> CreateTableAsync(CreateTableRequest request);
```

Parameters

Parameter	Description
tableMeta	The schema information of the table, which includes the following items: • tableName: the name of the base table. • primaryKeySchema: the schema of primary key for the table. For more information, see Primary keys and attributes. **Note** Attribute columns do not need to be defined. Different rows in Tablestore can have different attribute columns. The names of attribute columns can be specified when data is written. ◦ The primary key of a table can contain one to four primary key columns. Primary key columns are sorted in the order in which they are added. For example, PRIMARY KEY (A, B, C) and PRIMARY KEY (A, C, B) have different primary key schemas. Tablestore 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 stored in the same partition. Therefore, we recommend that you assign no more than 10 GB of data with the same partition key value. Otherwise, a single partition may be too large to split. We recommend that you set a well-designed primary key so that read and write operations can be evenly distributed to different partitions to facilitate load balancing. • definedColumnSchema: the predefined columns of the table and the data types of the predefined column values. Primary key columns cannot be set to predefined columns. You can use predefined columns as the indexed columns or attribute columns for index tables.

Parameter	Description
tableOptions	The configurations of the table. For more information, see Data versions and TTL. The configurations include the following items: • timeToLive: the period for which data in the table can be retained. This period is the validity period of data. When the retention period exceeds the timeToLive value, Tablestore automatically deletes expired data. The minimum timeToLive value is 86,400 (one day). A value of -1 indicates that data never expires. When you create a table, you can set timeToLive to -1 if you expect that data never expires. After the table is created, you can call the UpdateTable operation to modify the timeToLive value. Unit: seconds. **Note** You must set timeToLive to -1 if you want to create index tables for the base table. • maxVersions: the maximum number of versions of data that can be retained for a single attribute column. When the number of versions of data in attribute columns exceeds the maxVersions value, Tablestore automatically deletes data of earlier versions. When you create a table, you can customize the maximum number of versions that can be retained for the data in an attribute column. After the table is created, you can call the UpdateTable operation to modify the maxVersions value for the table. **Note** You must set maxVersions to 1 if you want to create index tables for the base table. • DeviationCellVersionInSec: the max version offset, which is the maximum difference between the timestamp of the written data and the current system time. The difference between the version number and the data written time must be within the value of DeviationCellVersionInSec. Otherwise, an error occurs when the data is written. Unit: seconds. The valid version range of data in an attribute column is calculated based on the formula: Valid version range = **[Data written time - Max version offset, Data written time + Max version offset)**. When you create a table, Tablestore uses the default value of 86400 if you do not specify a max version offset. After the table is created, you can call the UpdateTable operation to modify the DeviationCellVersionInSec value. Unit: seconds.

Parameter	Description
reservedThroughput	The reserved read and write throughput of the table. For capacity instances, the reserved read and write throughput can be set only to 0. Reserved throughput does not apply to these instances. The default value 0 indicates that all throughput is billed on a pay-as-you-go basis. Unit: CU. • When reserved read throughput or reserved write throughput is set to a value greater than 0, Tablestore allocates and reserves related resources for the table. After the table is created, Tablestore charges reserved throughput resources. Additional throughput is billed on a pay-as-you-go basis. For more information, see Billing overview. • When reserved read throughput or reserved write throughput is set to 0, Tablestore does not allocate or reserve related resources for the table.
indexMetas	The schema information of the index tables. Each indexMeta includes the following items: • indexName: the name of the index table. • primaryKey: the primary key of the index table, which is a combination of primary key columns and predefined columns of the base table. • definedColumns: the indexed attribute column, which is a combination of predefined columns of the base table. • indexUpdateMode: the update mode of the index table. Only IUM_ASYNC_INDEX is supported. • indexType: the type of the index table. Only IT_GLOBAL_INDEX is supported.

Examples

- Create a base table without creating an index table

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

```
 //Create a schema for the primary key columns, which includes the count, names, and type
s of primary key columns.
 //The first primary key column is named pk0 and requires an INTEGER value. The first pri
mary key column is also the partition key.
 //The second primary key column is named pk1 and requires an INTEGER value.
 var primaryKeySchema = new PrimaryKeySchema();
 primaryKeySchema.Add("pk0", ColumnValueType.Integer);
 primaryKeySchema.Add("pk1", ColumnValueType.String);
 //Create a tableMeta class based on the table name and the schema for primary key column
s.
 var tableMeta = new TableMeta("SampleTable", primaryKeySchema);
 // Set the reserved read throughput and reserved write throughput to 0.
 var reservedThroughput = new CapacityUnit(0, 0);
 try
 {
     // Construct the CreateTableRequest object.
     var request = new CreateTableRequest(tableMeta, reservedThroughput);
     // Call the CreateTable operation of the client. If the table is created, no excepti
on is returned. Otherwise, an exception is returned.
     otsClient.CreateTable(request);
     Console.WriteLine("Create table succeeded.") ;
 }
 # If the task fails, an exception is returned for you to handle.
 catch (Exception ex)
 {
     Console.WriteLine("Create table failed, exception:{0}", ex.Message);
 }
```

For the detailed sample code, visit CreateTable@GitHub.

- Create an index table when you create the base table

```
        public static void CreateTableWithGlobalIndex()
        {
            // Create a table that contains two primary key columns Pk1 and Pk2 and the p
redefined columns Col1 and col2.
            //Create a global secondary index. Specify Col1 as the first primary key colu
mn of the index table.
            OTSClient otsClient = Config.GetClient();
            Console.WriteLine("Start create table with globalIndex...") ;
            PrimaryKeySchema primaryKeySchema = new PrimaryKeySchema
                {
                    { Pk1, ColumnValueType.String },
                    { Pk2, ColumnValueType.String }
                };
            TableMeta tableMeta = new TableMeta(TableName, primaryKeySchema);
            tableMeta.DefinedColumnSchema = new DefinedColumnSchema {
                   { Col1, DefinedColumnType.STRING},
                   { Col2, DefinedColumnType.STRING}
                };
            IndexMeta indexMeta = new IndexMeta(IndexName);
            indexMeta.PrimaryKey = new List<string>() { Col1 };
            indexMeta.DefinedColumns = new List<string>() { Col2 };
            //indexMeta.IndexType = IndexType.IT_GLOBAL_INDEX;
            //indexMeta.IndexUpdateModel = IndexUpdateMode.IUM_ASYNC_INDEX;
            List<IndexMeta> indexMetas = new List<IndexMeta>() { };
            indexMetas.Add(indexMeta);
            CapacityUnit reservedThroughput = new CapacityUnit(0, 0);
            CreateTableRequest request = new CreateTableRequest(tableMeta, reservedThroug
hput, indexMetas);
            otsClient.CreateTable(request);
            Console.WriteLine("Table is created: " + TableName);
        }
```

6.4.2. UpdateTable

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

Operations

```
 /// <summary>
 /// Update the reserved read throughput or reserved write throughput of the specified tabl
e. New settings take effect within a minute after the update succeeds.
 /// </summary>
 /// <param name="request">Request parameter, including the table name and reserved read/wr
ite throughput.</param>
 /// <returns>Contains information such as the updated reserved read/write throughput.</ret
urns>
 public UpdateTableResponse UpdateTable(UpdateTableRequest request);
 /// <summary>
 /// Asynchronous mode of UpdateTable.
 /// </summary>
 public Task<UpdateTableResponse> UpdateTableAsync(UpdateTableRequest request);
```

Examples

Update the reserved read throughput to 1 and the reserved write throughput to 2.

```
 // Update the reserved read throughput to 1 and the reserved write throughput to 2.
 var reservedThroughput = new CapacityUnit(1, 2);
 // Construct an UpdateTableRequest object.
 var request = new UpdateTableRequest("SampleTable", reservedThroughput);
 try
 {
     // Call the operation to update the reserved read/write throughput of the table.
     otsClient.UpdateTable(request);
     // If the task succeeds, no exception is returned.
     Console.Writeline("Update table succeeded.");
 }
 catch (Exception ex)
 {
     // If the task fails, an exception is returned. Handle the exception.
     Console.WriteLine("Update table failed, exception:{0}", ex.Message);
 }
```

For the detailed sample code, visit UpdateTable@GitHub.

6.4.3. List the names of tables

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

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created.

API operations

```
/// <summary>
/// Query the names of all tables created in the current instance.
/// </summary>
/// <param name="request">Request parameter</param>
/// <returns>The table name list returned by the ListTable operation.</returns> </returns>
public ListTableResponse ListTable(ListTableRequest request);
/// <summary>
/// Asynchronous mode of ListTable.
/// </summary>
public Task<ListTableResponse> ListTableAsync(ListTableRequest request);
```

Examples

The following code provides an example on how to query 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);
}
```

6.4.4. Query the description of a table

> **Note** For more information about the DescribeTable operation, see DescribeTable.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created.

API operations

```
 /// <summary>
 /// Query the schema information and the reserved read throughput and reserved write throu
ghput of the specified table.
 /// </summary>
 /// <param name="request">Request parameters, including the table name.</param>
 /// <returns>Contains the schema information and the reserved read throughput and reserved
write throughput of the specified table. </returns>
 public DescribeTableResponse DescribeTable(DescribeTableRequest request);
 /// <summary>
 /// Asynchronous mode of DescribeTable.
 /// </summary>
 public Task<DescribeTableResponse> DescribeTableAsync(DescribeTableRequest request);
```

Parameters

Parameter	Description
tableName	The name of the table.

Examples

The following code provides an example on how to query the description of a table:

```
 try
 {
     var request = new DescribeTableRequest("SampleTable");
     var response = otsClient.DescribeTable(request);
     // Display the description of the specified table.
     Console.Writeline("Describe table succeeded.");
     Console.WriteLine("LastIncreaseTime: {0}", response.ReservedThroughputDetails.LastIncr
easeTime);
     Console.WriteLine("LastDecreaseTime: {0}", response.ReservedThroughputDetails.LastDecr
easeTime);
     Console.WriteLine("NumberOfDecreaseToday: {0}", response.ReservedThroughputDetails.Las
tIncreaseTime);
     Console.WriteLine("ReadCapacity: {0}", response.ReservedThroughputDetails.CapacityUnit
.Read);
     Console.WriteLine("WriteCapacity: {0}", response.ReservedThroughputDetails.CapacityUni
t.Write);
 }
 catch (Exception ex)
 {
     // If an exception is thrown, the operation fails and the system handles the exception
.
     Console.WriteLine("Describe table failed, exception:{0}", ex.Message);
 }
```

For the detailed sample code, visit DescribeTable@GitHub.

6.4.5. Delete tables

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

> **Note** For more information about the DeleteTable operation, see DeleteTable.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A table is created.
- The index tables and search indexes for the table are deleted.

Operations

```
/// <summary>
/// Delete a table based on the table name.
/// </summary>
/// <param name="request">Request parameter, including the table name.</param>
/// <returns>Information returned by DeleteTable. The result is null.
/// </returns>
public DeleteTableResponse DeleteTable(DeleteTableRequest request);
/// <summary>
/// Asynchronous mode of DeleteTable.
/// </summary>
public Task<DeleteTableResponse> DeleteTableAsync(DeleteTableRequest request);
```

Examples

The following code provides an example on how to delete a specified table:

```
var request = new DeleteTableRequest("SampleTable");
try
{
    otsClient.DeleteTable(request);
    Console.Writeline("Delete table succeeded.");
}
catch (Exception ex)
{
    Console.WriteLine("Delete table failed, exception:{0}", ex.Message);
}
```

For the complete sample code, visit DeleteTable@GitHub.

6.4.6. Configure an auto-increment primary key column

This topic describes how to configure an auto-increment primary key column. You cannot set a partition key to an auto-increment column. If you write data to a table that contains an auto-increment primary key column, you do not need to specify specific values for the auto-increment primary key column because Tablestore generates values for the auto-increment primary key column. Values generated for the auto-increment primary key column are unique and consecutive within the partition that shares the same partition key value.

Prerequisites

Client is initialized. For more information, see Initialization.

Usage notes

1. When you create a table, you cannot set the partition key to an auto-increment primary key column.

   The data type of an auto-increment primary key column can only be set to integer. Each value generated for an auto-increment primary key column is a 64-bit signed long integer.

2. When you write data to a table, you do not need to set specific values for the auto-increment column. You need only to set placeholder values for the auto-increment column.

   If you want to obtain the values of the auto-increment column after data is written to the table, you can set ReturnType to RT_PK.

   When you query data, you must specify the values of all primary key columns. To obtain a complete primary key value, you can set ReturnType to RT_PK in PutRow, UpdateRow, or BatchWriteRow.

Examples

When you use auto-increment primary key columns, you can call the CreateTable, PutRow, UpdateRow, or BatchWriteRow operation.

- To create an auto-increment primary key column when you create a table, you must set the attribute of the primary key column to AUTO_INCREMENT.
- When you write data to a table, you do not need to set specific values for the auto-increment primary key column. You need only to set placeholder values for the column.

```
using System;
using System.Collections.Generic;
using Aliyun.OTS.DataModel;
using Aliyun.OTS.Request;
using Aliyun.OTS.Response;
namespace Aliyun.OTS.Samples.Samples
{
    /// <summary>
    /// Example: Create an auto-increment primary key column.
    /// </summary>
    public class AutoIncrementSample
    {
        private static readonly string TableName = "AutoIncrementSample";
        private static readonly string Pk1 = "Pk1";
        private static readonly string Pk2 = "Pk2_AutoIncrement";
        static void Main(string[] args)
        {
            Console.WriteLine("AutoIncrementSample");
            // Create a table that contains an auto-increment column.
```

```
            CreateTableWithAutoIncrementPk();
            // Write 10 rows of data.
            for (int i = 0; i < 10; i++)
            {
                PutRow(i.ToString());
            }
            Console.ReadLine();
        }
        /// <summary>
        /// Create a table that contains an auto-increment column.
        /// </summary>
        private static void CreateTableWithAutoIncrementPk()
        {
            OTSClient otsClient = Config.GetClient();
            IList<string> tables = otsClient.ListTable(new ListTableRequest()).TableNames;
            if (tables.Contains(TableName))
            {
                return;
            }
            PrimaryKeySchema primaryKeySchema = new PrimaryKeySchema
            {
                { Pk1, ColumnValueType.String },
                // Set Pk2 to the auto-increment column.
                { Pk2, ColumnValueType.Integer, PrimaryKeyOption.AUTO_INCREMENT}
            };
            TableMeta tableMeta = new TableMeta(TableName, primaryKeySchema);
            CapacityUnit reservedThroughput = new CapacityUnit(0, 0);
            CreateTableRequest request = new CreateTableRequest(tableMeta, reservedThroughp
ut);
            otsClient.CreateTable(request);
        }
        public static void PutRow(string pk1Value)
        {
            Console.WriteLine("Start put row...") ;
            OTSClient otsClient = Config.GetClient();
            // Specify the primary keys of the row. These primary keys must be consistent w
ith those specified in TableMeta when you create the table.
            PrimaryKey primaryKey = new PrimaryKey
            {
                { Pk1, new ColumnValue(pk1Value) },
                { Pk2,  ColumnValue.AUTO_INCREMENT }
            };
            // Specify the attribute columns of the row.
            AttributeColumns attribute = new AttributeColumns
            {
                { "Col1", new ColumnValue(0) }
            };
            PutRowRequest request = new PutRowRequest(TableName, new Condition(RowExistence
Expectation.IGNORE), primaryKey, attribute);
            request.RowPutChange.ReturnType = ReturnType.RT_PK;
            var response =  otsClient.PutRow(request);
            Console.WriteLine("Put row succeed, autoIncrement Pk value:"+ response.Row.GetP
rimaryKey()[Pk2].IntegerValue);
        }
```

```
        }
    }
```

6.4.7. Configure conditional update

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

When you use the PutRow, UpdateRow, DeleteRow, or BatchWriteRow operation to update data, you can use conditional update to check row existence conditions and column-based conditions. The update is successful only when the conditions are met.

Conditional update includes row existence conditions and column-based conditions.

- Row existence conditions: include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST.

  When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported.

- Column-based conditions: include RelationalCondition and CompositeCondition, which are used to determine whether the conditions are met based on the values of one or more columns.
  - RelationalCondition supports the comparison between a constant and a column that can be a primary key column. RelationalCondition does not support the comparison between two columns or two constants.
  - The inner node of CompositeCondition is used to perform logical operations. Subconditions can be RelationalCondition or CompositeCondition.

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

Parameters

Parameter	Description
RowExistenceExpectation	When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported. Row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST. • IGNORE: No existence check is performed. • EXPECT_EXIST: The row is expected to exist. If the row exists, the condition is met. If the row does not exist, the condition is not met. • EXPECT_NOT_EXIST: Expect that the row does not exist. If the row does not exist, the condition is met. If the row exists, the condition is not met.
ColumnName	The column name.

Parameter	Description
ColumnValue	The comparison value of the column.
Operator	The relational operator that is used to compare column values. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_THAN (>=), LESS_THAN (<), and LESS_EQUAL (<=).
LogicOperator	The logical operator that is used to combine multiple conditions. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. Different logical operators require different number of subconditions. • When the logical operator is NOT, only one subcondition can be added. • When the logical operator is AND or OR, you must add at least two subconditions.
PassIfMissing	Specifies whether to pass the conditional check when a column does not exist in a row. The type of this parameter value is Boolean. The default value is true, which indicates that if the column does not exist in a row, the conditional check is passed, and the row meets the update conditions. When PassIfMissing is set to false, if the column does not exist in a row, the conditional check fails, and the row does not meet the update conditions.
LatestVersionsOnly	Specifies whether to use only the value of the latest version when the comparison value has multiple versions. The type of this parameter value is Boolean. The default value is true, which indicates that if the column has multiple versions of data, only the value of the latest version is used for comparison. When LatestVersionsOnly is set to false, the comparison values of all versions are used for comparison if the comparison value has multiple versions. In this case, if only the value of a version meets the condition, the conditional check is passed, and the row meets the update conditions.

Examples

```
    // Specify the primary key of the row. The primary key must be consistent with the prim
ary key specified in TableMeta when you create a table.
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    // Specify the attribute columns of the row.
    AttributeColumns attribute = new AttributeColumns();
    attribute.Add("col0", new ColumnValue(0));
    attribute.Add("col1", new ColumnValue("a"));
    attribute.Add("col2", new ColumnValue(true));
    PutRowRequest request = new PutRowRequest(tableName, new Condition(RowExistenceExpectat
ion.IGNORE), primaryKey, attribute);
    // Call the PutRow operation when other conditions are not configured. The operation sug
```

```
    // Call the PutRow operation when other conditions are not configured. The operation succeeds.
    try
    {
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    } catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
    // When the value of col0 is not equal to 5, call the PutRow operation again to overwrite the original value. The operation succeeds.
    try
    {
        request.Condition.ColumnCondition = new RelationalCondition("col0",
                                            CompareOperator.NOT_EQUAL,
                                            new ColumnValue(5));
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    } catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
    // When the value of col0 is equal to 5, call the PutRow operation again to overwrite the original value. The operation fails.
    try
    {
        // Add a new condition: The value of col0 is equal to 5.
        request.Condition.ColumnCondition = new RelationalCondition("col0",
                                            CompareOperator.EQUAL,
                                            new ColumnValue(5));
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    }
    catch (OTSServerException)
    {
        // OTSServerException is returned because the condition is not met.
        Console.WriteLine("Put row failed  because condition check failed. but expected");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
```

6.4.8. Configure atomic counter

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

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Limits

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

API operations

The API operations to implement the atomic counter feature are added to the rowUpdateChange class. The following table lists these operations.

Operation	Description
RowUpdateChange Increment(Column column)	Increases or decreases a value by a number.
List<String> ReturnColumnNames	Specifies the column names whose value needs to be returned for the columns on which operations are performed to implement atomic counter.
ReturnType ReturnType	Specifies the return type. The updated value in the column on which operations are performed to implement atomic counter must be returned.

Parameters

Parameter	Description
TableName	The name of the table.
ColumnName	The name of the column on which operations are performed to implement atomic counter. Columns whose valid values are integers are supported to implement the atomic counter feature.
Value	The value to update the columns on which operations are performed to implement atomic counter.

Parameter	Description
ReturnColumnNames	Specifies the column names whose value needs to be returned for the columns on which operations are performed to implement atomic counter.
ReturnType	Sets the return type to ReturnType.RT_AFTER_MODIFY and returns the values of the columns on which operations are performed to implement atomic counter.

Examples

The following code provides an example on how to call the rowUpdateChange operation to update the columns whose valid values are integers and return the updated values when you write a row of data:

```
public static void Increment(int incrementValue)
{
    Console.WriteLine("Start set increment column...") ;
    OTSClient otsClient = Config.GetClient();
    // Specify the primary key of the row. The primary key must be consistent with the prim
ary key specified in TableMeta when you create a table.
    PrimaryKey primaryKey = new PrimaryKey
    {
        { Pk1, new ColumnValue(0) },
        { Pk2, new ColumnValue("abc") }
    };
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TableName, primaryKey); // Specif
y the name of the table.
    // Set the return type to ReturnType.RT_AFTER_MODIFY and return the values of the colum
ns on which operations are performed to implement atomic counter.
    rowUpdateChange.ReturnType = ReturnType.RT_AFTER_MODIFY;
    rowUpdateChange.ReturnColumnNames = new List<string>() { IncrementCol};
    // Set the column on which operations are performed to implement atomic counter. The co
lumn value starts from 0 and increases by the number specified by the incrementValue parame
ter.
    rowUpdateChange.Increment(new Column(IncrementCol, new ColumnValue(incrementValue)));
    UpdateRowRequest updateRowRequest = new UpdateRowRequest(rowUpdateChange);
    var response = otsClient.UpdateRow(updateRowRequest);
    Console.WriteLine("set Increment column succeed, Increment result:" + response.Row.GetC
olumns()[0].Value);
}
```

6.4.9. Configure filter

Tablestore filters query results on the server side before returning the results. Only rows of data that match the filter conditions are returned.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

When you query data by calling the GetRow, BatchGetRow, or GetRange operation, you can use the filter to return only the rows that meet the filter conditions.

Filters include RelationalCondition and CompositeCondition.

- RelationalCondition: determines whether to filter a row based only on the values of a reference column.
- CompositeCondition: determines whether to filter a row by combining the filter conditions of values of multiple reference columns.

Limit

- The filter conditions support relational operators such as =, !=, >, >=, <, and <=, and logical operators such as NOT, AND, and OR. You can specify a maximum of 10 conditions.
- Reference columns in filters must be included in the read result. If the specified columns from which data is read do not include reference columns, the filter cannot query the values of reference columns.
- You can use filters by calling the GetRow, BatchGetRow, and GetRange operations, which does not change the native semantics or limited items of these operations.

  When you use GetRange, the number of rows scanned cannot exceed 5,000, or the data scanned cannot exceed 4 MB in size.

  If the scanned 5,000 rows or 4 MB does not match the filter conditions, the rows in the response are empty. However, nextStartPrimaryKey may not be empty. In this case, you need to use nextStartPrimaryKey to continue reading the data until nextStartPrimaryKey is empty.

Parameters

Parameter	Description
ColumnName	The name of the reference column in the filter.
ColumnValue	The comparison value of the reference column in the filter.
ComparatorType	The relational operator in the filter. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=).
LogicOperator	The logical operator in the filter. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR.
PassIfMissing	Specifies whether to return a row if the row does not contain the reference column. The type of this parameter value is Boolean. The default value is true, which indicates that if the reference column does not exist in a row, the row is returned. When PassIfMissing is set to false, if the reference column does not exist in a row, the row is not returned.

Parameter	Description
LatestVersionsOnly	Specifies whether to use only the value of the latest version for comparison when the reference column contains data of multiple versions. The type of this parameter value is Boolean. The default value is true, which indicates that if the reference column has multiple versions of data, only the value of the latest version is used for comparison. When LatestVersionsOnly is set to false, if a reference column has multiple versions of data, the values of all versions in the column are used for comparison. In this case, if only the value of one version meets the conditions, the row is returned.

Examples

- The following code provides an example on how to construct RelationalCondition:

```
public void GetRowWithRelationalCondition(OTSClient otsClient)
{
    // Specify the primary key of the row. The primary key must be consistent with the pr
imary key specified in TableMeta when you create a table.
    PrimaryKey primaryKey = new PrimaryKey
    {
        { "pk0", new ColumnValue(0) },
        { "pk1", new ColumnValue("abc") }
    };
    var rowQueryCriteria = new SingleRowQueryCriteria(TableName)
    {
        RowPrimaryKey = primaryKey
    };
    // Specify that the rows are returned when the value of col0 is 5.
    var filter = new RelationalCondition("col0",CompareOperator.EQUAL,new ColumnValue(5))
    {
        PassIfMissing = true
    };
    rowQueryCriteria.Filter = filter.ToFilter();
    rowQueryCriteria.AddColumnsToGet("col0");
    rowQueryCriteria.AddColumnsToGet("col1");
    GetRowRequest request = new GetRowRequest(rowQueryCriteria);
    // Execute the query.
    GetRowResponse response = otsClient.GetRow(request);
    PrimaryKey primaryKeyRead = response.PrimaryKey;
    AttributeColumns attributesRead = response.Attribute;
    Console.WriteLine("Primary key read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in primaryKeyRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Attributes read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in attributesRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Get row with filter succeed.") ;
}
```

- The following code provides an example on how to construct CompositeCondition:

```
public void GetRowWithCompositeCondition(OTSClient otsClient)
{
    // Specify the primary key of the row. The primary key must be consistent with the pr
imary key specified in TableMeta when you create a table.
    PrimaryKey primaryKey = new PrimaryKey
    {
        { "pk0", new ColumnValue(0) },
        { "pk1", new ColumnValue("abc") }
    };
    var rowQueryCriteria = new SingleRowQueryCriteria(TableName)
    {
        RowPrimaryKey = primaryKey
        };
    // Specify that the rows are returned when the value of col0 is 5, or the value of co
l1 is not ff.
    var filter1 = new RelationalCondition("col0",
                                          CompareOperator.EQUAL,
                                          new ColumnValue(5));
    var filter2 = new RelationalCondition("col1", CompareOperator.NOT_EQUAL, new ColumnVa
lue("ff"));
    var filter = new CompositeCondition(LogicOperator.OR);
    filter.AddCondition(filter1);
    filter.AddCondition(filter2);
    rowQueryCriteria.Filter = filter.ToFilter();
    rowQueryCriteria.AddColumnsToGet("col0");
    rowQueryCriteria.AddColumnsToGet("col1");
    GetRowRequest request = new GetRowRequest(rowQueryCriteria);
    // Execute the query.
    GetRowResponse response = otsClient.GetRow(request);
    PrimaryKey primaryKeyRead = response.PrimaryKey;
    AttributeColumns attributesRead = response.Attribute;
    Console.WriteLine("Primary key read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in primaryKeyRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Attributes read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in attributesRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Get row with filter succeed.") ;
}
```

6.5. Single-row operations

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

PutRow

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

- Operations

```
/// <summary>
/// Write a row of data based on the specified table name, primary keys, and attributes.
Thu number of capacity units (CUs) consumed by the operation is returned.
/// </summary>
/// <param name="request">Data insertion request</param>
/// <returns>The number of CUs consumed by the operation</returns>
public PutRowResponse PutRow(PutRowRequest request);
/// <summary>
/// The asynchronous mode of PutRow.
/// </summary>
public Task<PutRowResponse> PutRowAsync(PutRowRequest request);
```

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** ◦ The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table. ◦ For an auto-increment primary key column, you need only to set the value of the auto-increment primary key column to a placeholder. For more information, see Configure an auto-increment primary key column.

Parameter	Description
attribute	The attribute columns of the row. ○ Each attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value type (optional), attribute column value, and timestamp (optional). ○ A timestamp is the data version number. For more information, see Data versions and TTL. You can customize a data version number or use the data version number generated by Tablestore. If you do not specify this parameter, the data version number generated by the system is used. ■ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ■ When you choose to specify the version number, ensure that the version number is a 64-bit timestamp accurate to the millisecond within the valid version range.
condition	You can use conditional update to set row existence conditions or column-based conditions. For more information, see Configure conditional update. **Note** ○ In addition to row conditions, the Condition parameter also supports column-based conditions in Tablestore SDK for .NET 2.2.0 and later. ○ Condition.IGNORE, Condition.EXPECT_EXIST, and Condition.EXPECT_NOT_EXIST are deprecated since Tablestore SDK for .NET 3.0.0. Replace them with new Condition (RowExistenceExpectation.IGNORE), new Condition (RowExistenceExpectation.EXPECT_EXIST), and new Condition (RowExistenceExpectation.EXPECT_NOT_EXIST). ○ RowExistenceExpectation.IGNORE indicates that new data is inserted into a row regardless of whether the specified row exists. If the specified row exists, the existing data is overwritten. ○ RowExistenceExpectation.EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten. ○ RowExistenceExpectation.EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist.

- Example 1:

  The following code provides an example on how to insert a row of data:

```
    // Specify the primary key of the row. The primary key must be consistent with the pr
imary key specified in TableMeta when you create a table.
    var primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    // Specify the attribute columns of the row.
    var attribute = new AttributeColumns();
    attribute.Add("col0", new ColumnValue(0));
    attribute.Add("col1", new ColumnValue("a"));
    attribute.Add("col2", new ColumnValue(true));
    try
    {
        // Construct the request object to insert a row of data. RowExistenceExpectation.
IGNORE indicates that data is inserted no matter whether the specified row exists.
        var request = new PutRowRequest("SampleTable", new Condition(RowExistenceExpectat
ion.IGNORE),
                                    primaryKey, attribute);
        // Call the PutRow operation to insert data.
        otsClient.PutRow(request);
        // If the operation succeeds, no exception is returned.
        Console.WriteLine("Put row succeeded.");
    }
    catch (Exception ex)
    {
        // If the operation fails, an exception is returned and handled.
        Console.WriteLine("Put row failed, exception:{0}", ex.Message);
    }
```

  For the detailed sample code, visit PutRow@GitHub.

- Example 2

  The following code provides an example on how to insert a row of data based on specified conditions. In the following example, the data is inserted only when the row exists and the value of col0 is greater than 24.

```
    // Specify the primary key of the row. The primary key must be consistent with the pr
imary key specified in TableMeta when you create a table.
    var primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    // Specify the attribute columns of the row.
    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(RowExistenceExpectat
ion.EXPECT_EXIST),
                                primaryKey, attribute);
    // When the value of col0 is greater than 24, the row can be inserted to overwrite th
e original value.
    try
    {
        request.Condition.ColumnCondition = new RelationalCondition("col0",
                                            RelationalCondition.CompareOperator.GREATER_T
HAN,
                                            new ColumnValue(24));
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
```

For the detailed sample code, visit ConditionPutRow@GitHub.

- Example 3

  The following code provides an example on how to asynchronously insert a row of data.

  > **Notice** Each asynchronous call starts a thread. A timeout error may occur if too many asynchronous invocations are started consecutively and each invocation consumes an extended period of time.

```
    try
    {
        var putRowTaskList = new List<Task<PutRowResponse>>();
        for (int i = 0; i < 100; i++)
        {
            // Specify the primary key of the row. The primary key must be consistent wit
h the primary key specified in TableMeta when you create a table.
            var primaryKey = new PrimaryKey();
            primaryKey.Add("pk0", new ColumnValue(i));
            primaryKey.Add("pk1", new ColumnValue("abc"));
            // Specify the attribute columns of the row.
            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(RowExistenceExpectat
ion.IGNORE),
                                            primaryKey, attribute);
            putRowTaskList.Add(TabeStoreClient.PutRowAsync(request));
        }
        // Wait until each asynchronous call returns results and displays the consumed CU
s.
        foreach (var task in putRowTaskList)
        {
            task.Wait();
            Console.WriteLine("consumed read:{0}, write:{1}", task.Result.ConsumedCapacit
yUnit.Read,
                                task.Result.ConsumedCapacityUnit.Write);
        }
        // If the operation succeeds, no exception is returned.
        Console.WriteLine("Put row async succeeded.");
    }
    catch (Exception ex)
    {
        // If the operation fails, an exception is returned and handled.
        Console.WriteLine("Put row async failed. exception:{0}", ex.Message);
    }
```

For the detailed sample code, visit PutRowAsync@GitHub.

GetRow

You can call this operation to read a row of data.

The following results of the read request may be returned:

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

```
/// <summary>
/// Read a single row of data based on a specified primary key.
/// </summary>
/// <param name="request">Data query request</param>
/// <returns>Response of GetRow</returns>
public GetRowResponse GetRow(GetRowRequest request);
/// <summary>
/// The asynchronous mode of GetRow.
/// </summary>
public Task<GetRowResponse> GetRowAsync(GetRowRequest request);
```

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
columnsToGet	The names of the columns to return. You can specify the names of the primary key columns and the names of attribute columns. If you do not specify this parameter, all data in the row is returned. **Note** ◦ By default, Tablestore returns the data from all columns of the row when you query a row. You can use the columnsToGet parameter to read data from specified columns. For example, the system returns only the values of col0 and col1 if col0 and col1 are included in columnsToGet. ◦ If you use columnsToGet and filter together, the system first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.

Parameter	Description
maxVersions	The maximum number of versions that can be read. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data whose versions are within the specified time range or data of the specified version is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest.
timeRange	Specifies a range of versions to read or a version of data to read. For more information, see TimeRange. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data whose versions are within the specified time range or data of the specified version is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ To query data within a range, you must set StartTime and EndTime. StartTime specifies the start timestamp. EndTime specifies the end timestamp. The time range is a left-closed and right-open interval, which is [start_time, end_time). ◦ To query data within a range of versions, set SpecificTime. SpecificTime specifies a specific timestamp. Only one of SpecificTime and [StartTime, EndTime) is required. The timestamp used for the value of time_range ranges from 0 to Int64.MaxValue. Unit: milliseconds.

Parameter	Description
filter	You can set filter conditions to filter the queried results on the server side. Only rows that meet the specified filter conditions are returned. For more information, see Configure filter. **Note** If you use columnsToGet and filter together, the system first queries the columns specified by columnsToGet, and then returns rows that meet the filter conditions.

- Example 1:

  The following code provides an example on how to read a row of data:

```
    // Specify the primary key of the row. The primary key must be consistent with the pr
imary key specified in TableMeta when you create a table.
    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 speci
fied.
        var request = new GetRowRequest(TableName, primaryKey);
        // Call the GetRow operation to query data.
        var response = otsClient.GetRow(request);
        // Generate the data of the row. The code for generated data is omitted. For more
information, visit the GitHub link of the sample code.
        // If the operation succeeds, no exception is returned.
        Console.WriteLine("Get row succeeded.");
    }
    catch (Exception ex)
    {
        // If the operation fails, an exception is returned and handled.
        Console.WriteLine("Update table failed, exception:{0}", ex.Message);
    }
```

  For the detailed sample code, visit GetRow@GitHub.

- Example 2

  The following code provides an example on how to use the filter feature to read a row of data.

  In this example, the values of col0 and col1 that meet the following filter condition are set to returned: the value of col0 is 5 or the value of col1 is not ff.

```
    // Specify the primary key of the row. The primary key must be consistent with the pr
imary key specified in TableMeta when you create a table.
    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 equal to ff.
    var filter2 = new RelationalCondition("col1", RelationalCondition.CompareOperator.NOT
_EQUAL, new ColumnValue("ff"));
    // Construct a combination of Condition 1 and Condition 2. The relationship of the co
mbination is OR.
    var filter = new CompositeCondition(CompositeCondition.LogicOperator.OR);
    filter.AddCondition(filter1);
    filter.AddCondition(filter2);
    rowQueryCriteria.Filter = filter;
    // Set to return the values of col0 and col1 from the rows you want to query. Then, f
ilter the queried data based on the specified conditions.
    rowQueryCriteria.AddColumnsToGet("col0");
    rowQueryCriteria.AddColumnsToGet("col1");
    // Construct GetRowRequest.
    var request = new GetRowRequest(rowQueryCriteria);
    try
    {
        // Perform the query.
        var response = otsClient.GetRow(request);
        // Generate data or perform the related logical operation. The code for generated
data is omitted.
        // If the operation succeeds, no exception is returned.
        Console.WriteLine("Get row with filter succeeded.");
    }
    catch (Exception ex)
    {
        // If the operation fails, an exception is returned and handled.
        Console.WriteLine("Get row with filter failed, exception:{0}", ex.Message);
    }
```

For the detailed sample code, visit GetRowWithFilter@GitHub.

UpdateRow

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

> **Note** If the UpdateRow request contains only columns to delete, and the specified row does not exist, a new row is not added.

- Operations

```
    /// <summary>
    /// You can call this operation to update the data of a specified row. If the row doe
s not exist, a new row is added. If the row exists, the values of the specified columns a
re added, modified, or deleted based on the request content.
    /// </summary>
    /// <param name="request">Request instance</param>
    public UpdateRowResponse UpdateRow(UpdateRowRequest request);
    /// <summary>
    /// The asynchronous mode of UpdateRow.
    /// </summary>
    /// <param name="request"></param>
    /// <returns></returns>
    public Task<UpdateRowResponse> UpdateRowAsync(UpdateRowRequest request);
```

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.
attribute	The attribute column to be updated.

- Examples

  The following code provides an example on how to update the data of a specified row:

```
    // Specify the primary key of the row. The primary key must be consistent with the pr
imary key specified in TableMeta when you create a table.
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    // Specify the attribute columns of the row.
    UpdateOfAttribute attribute = new UpdateOfAttribute();
    attribute.AddAttributeColumnToPut("col0", new ColumnValue(0));
    attribute.AddAttributeColumnToPut("col1", new ColumnValue("b")); // Set the original
value of 'a' to 'b'.
    attribute.AddAttributeColumnToPut("col2", new ColumnValue(true));
    try
    {
        // Construct the request object to update the row. RowExistenceExpectation.IGNORE
indicates that data is updated no matter whether the specified row exists.
        var request = new UpdateRowRequest(TableName, new Condition(RowExistenceExpectati
on.IGNORE),
                                    primaryKey, attribute);
        // Call the UpdateRow operation to update data.
        otsClient.UpdateRow(request);
        // If the operation succeeds, no exception is returned.
        Console.Writeline("Update row succeeded.");
    }
    catch (Exception ex)
    {
        // If the operation fails, an exception is returned and handled.
        Console.WriteLine("Update row failed, exception:{0}", ex.Message);
    }
```

For the detailed sample code, visit UpdateRow@GitHub.

DeleteRow

You can call this operation to delete a row. If the specified row does not exist, no change is made to the table.

- Operations

```
    /// <summary>
    /// Delete a row of data based on the specified table name and primary keys of the ro
w.
    /// </summary>
    /// <param name="request">Request instance</param>
    /// <returns>Response instance</returns>
    public DeleteRowResponse DeleteRow(DeleteRowRequest request);
    /// <summary>
    /// The asynchronous mode of DeleteRow.
    /// </summary>
    public Task<DeleteRowResponse> DeleteRowAsync(DeleteRowRequest request);
```

- Parameters

Parameter	Description
tableName	The name of the table.
primaryKey	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.

- Examples

  The following code provides an example on how to delete a row from a table:

```
    // 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. RowExistenceExpectation.EXPECT_EXIST indicates that the r
ow is deleted only when the specified row exists.
        var deleteRowRequest = new DeleteRowRequest("SampleTable", new Condition(RowExist
enceExpectation.EXPECT_EXIST), primaryKey);
        // Call the DeleteRow operation to delete data.
        otsClient.DeleteRow(deleteRowRequest);
        // If the operation succeeds, no exception is returned.
        Console.Writeline("Delete table succeeded.");
    }
    catch (Exception ex)
    {
        // If the operation fails, an exception is returned and handled.
        Console.WriteLine("Delete table failed, exception:{0}", ex.Message);
    }
```

  For the detailed sample code, visit DeleteRow@GitHub.

6.6. Multi-row operations

Tablestore SDK for .NET provides multi-row operations such as BatchGetRow, BatchWriteRow, GetRange, and GetRangeIterator.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

BatchWriteRow

You can call this operation to write several rows of data to one or more tables by sending one request. The BatchWriteRow operation consists of PutRow, UpdateRow, and DeleteRow operations. The process of constructing a suboperation is the same as that of calling the PutRow, UpdateRow, and DeleteRow operations. BatchWriteRow supports conditional updates.

The execution and returning of results of each suboperation are carried out independently.

When multiple rows are written, part of rows may fail to be written. If the operation fails for part of rows, the system does not return exceptions. However, information about the index and error messages of the failed rows is contained in BatchWriteRowResponse. Therefore, when you call the BatchWriteRow operation, you must check the return values and check whether the status of each row is successful. If you do not check the return values, failures of operations on part of rows are ignored.

When the server detects errors caused by operations, the BatchWriteRow operation may return exceptions of parameter errors. All operations for the request are not executed.

- Operations

```
/// <summary>
/// <para>Insert, modify, or delete several rows of data in one or more tables. </para>
/// <para>The BatchWriteRow operation is a set of multiple PutRow, UpdateRow, and DeleteR
ow operations. The execution, returning of results, and capacity unit (CU) consumption of
each individual operation are carried out independently. </para>
/// <para>Compared with the execution of a large number of write operations, the use of B
atchWriteRow can reduce the request response time and increase the data write rate. </par
a>
/// </summary>
/// <param name="request">Request instance</param>
/// <returns>Response instance</returns>
public BatchWriteRowResponse BatchWriteRow(BatchWriteRowRequest request);
/// <summary>
/// Asynchronous mode of BatchWriteRow.
/// </summary>
/// <param name="request"></param>
/// <returns></returns>
public Task<BatchWriteRowResponse> BatchWriteRowAsync(BatchWriteRowRequest request);
```

- Parameters

  For information about parameters, see Single-row operations.

- Examples

  The following code provides an example on how to write 100 rows of data:

```
// Construct a request object to write multiple rows of data. Set the primary key for 100
rows of data.
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"));
    // Specify the attribute columns of the row.
    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, attri
bute);
}
request.Add(TableName, rowChanges);
try
{
    // Call the BatchWriteRow operation.
    var response = otsClient.BatchWriteRow(request);
    var tableRows = response.TableRespones;
    var rows = tableRows[TableName];
    // When multiple rows are written, part of rows may fail to be read. You must check t
he return values and check whether the status of each row is successful. For more informa
tion, see the link of GitHub in the sample code.
}
catch (Exception ex)
{
    // If the task fails, an exception is returned. Handle the exception.
    Console.WriteLine("Batch put row failed, exception:{0}", ex.Message);
}
```

For the detailed sample code, visit BatchWriteRow@GitHub.

BatchGetRow

You can call this operation to read several rows of data from one or more tables by sending one request. The BatchGetRow operation consists of multiple GetRow operations. The process of constructing a suboperation is the same as that of calling the GetRow operation. BatchGetRow supports filters.

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

The execution and returning of results of each suboperation are carried out independently.

When multiple rows are read, part of rows may fail to be read. If the operation fails for part of rows, the system does not return exceptions. However, information about the error messages of the failed rows is contained in BatchGetRowResponse. Therefore, when you call the BatchGetRow operation, you must check the return values and check whether the status of each row is successful.

- Operations

```
/// <summary>
/// <para>Simultaneously read several rows of data from one or more tables. </para>
/// <para>The BatchGetRow operation is a set of multiple GetRow operations. The execution
, returning of results, and CU consumption of each individual operation are carried out i
ndependently. </para>
/// Compared with the execution of a large number of GetRow operations, the use of BatchG
etRow can reduce the request response time and increase the data read rate.
/// </summary>
/// <param name="request">Request instance</param>
/// <returns>Response instance</returns>
public BatchGetRowResponse BatchGetRow(BatchGetRowRequest request);
/// <summary>
/// Asynchronous mode of BatchGetRow.
/// </summary>
public Task<BatchGetRowResponse> BatchGetRowAsync(BatchGetRowRequest request);
```

- Parameters

  For information about parameters, see Single-row operations.

- Examples

  The following code provides an example on how to read 10 rows of data at a time:

```
// Construct a request object to read multiple rows of data. Set the primary key for 10 r
ows of data.
List<PrimaryKey> primaryKeys = new List<PrimaryKey>();
for (int i = 0; i < 10; i++)
{
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(i));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    primaryKeys.Add(primaryKey);
}
try
{
    BatchGetRowRequest request = new BatchGetRowRequest();
    request.Add(TableName, primaryKeys);
    // Call BatchGetRow to read 10 rows of data.
    var response = otsClient.BatchGetRow(request);
    var tableRows = response.RowDataGroupByTable;
    var rows = tableRows[TableName];
    // Export data in rows. The code for this operation is ignored. For more information,
see the link of GitHub in the sample code.
    // When multiple rows are written, part of rows may fail to be read. You must check t
he return values and check whether the status of each row is successful. For more informa
tion, see the link of GitHub in the sample code.
}
catch (Exception ex)
{
    // If the task fails, an exception is returned. Handle the exception.
    Console.WriteLine("Batch get row failed, exception:{0}", ex.Message);
}
```

  For the detailed sample code, visit BatchGetRow@GitHub.

GetRange

You can call this operation to read rows of data in a forward or backward direction based on a specified primary key range.

You can also set the number of rows you want 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 the next primary key information are returned. You can also initiate a request to start from where the last operation left off and read the remaining rows based on the next primary key information returned by the previous operation.

The GetRange operation may stop and return data in the following situations:

- The amount of data read reaches 4 MB.
- The number of rows read reaches 5,000.
- The number of rows returned reaches the limit.
- All reserved read throughput is consumed. You have no reserved read throughput to read the next row of data.

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

- Operations

```
/// <summary>
/// Obtain data from multiple rows within a specified range.
/// </summary>
/// <param name="request">Request instance</param>
/// <returns>Response instance</returns>
public GetRangeResponse GetRange(GetRangeRequest request);
/// <summary>
/// Asynchronous mode of GetRange.
/// </summary>
/// <param name="request"></param>
/// <returns></returns>
public Task<GetRangeResponse> GetRangeAsync(GetRangeRequest request);
```

- Parameters

Parameter	Description
tableName	The name of the table.

Parameter	Description
direction	The direction for reading. ◦ If the direction is set to FORWARD, the start primary key column value must be smaller than the end primary key column value. The rows are returned based on the primary key column values in ascending order. ◦ If the direction is set to BACKWARD, the start primary key column value must be greater than the end primary key column value. The rows are returned based on the primary key column values in descending order. Example: A table has two primary keys A and B where the value of A is smaller than that of B. If you set direction to forward, the rows whose primary key column values are greater than or equal to the value of A and smaller than the value of B are returned in ascending order. If you set direction to backward, the rows whose primary key column values are greater than the value of A and smaller than or equal to the value of B are returned in descending order.
inclusiveStartPrimaryKey exclusiveEndPrimaryKey	The start and end primary keys of the read operation. The start and end primary keys must be valid primary keys or virtual points of the INF_MIN or INF_MAX type. The number of columns for the virtual point must be the same as that of the primary key. INF_MIN indicates an infinitely small value of which all other values of other types are greater. INF_MAX indicates an infinitely large value of which all other values are smaller. ◦ inclusiveStartPrimaryKey specifies the start primary key. If the row exists, the response contains this row. ◦ exclusiveEndPrimaryKey specifies the end primary key. The response excludes the row no matter whether the row exists. Rows in tables are sorted based on the primary key column values in ascending order. The range for reading is a left-closed and right-open interval. When data is read in the forward direction, all rows whose primary key column values are greater than or equal to the start primary key column value and smaller than the end primary key column value are returned.
limit	The maximum number of returned rows. This parameter value must be greater than 0. An operation stops when the maximum number of rows is returned in the forward or backward direction, even if part of rows within the specified range are not returned. At this time, you can use the checkpoint value in the response for the next reading.

Parameter	Description
columnsToGet	The collection of columns you want to read. The columns can be primary key columns or attribute columns. If this parameter is not set, rows that contain all columns are returned. **Note** ◦ If you query a row of data, the system returns the data in all columns of the row. You can use the columnsToGet parameter to read data from specified columns. If col0 and col1 are added to columnsToGet, only the values of the col0 and col1 columns are returned. ◦ If the primary key set for a row is within the specified range, but the row does not contain the specified columns, the response does not contain the row of data. ◦ If columnsToGet and filter are used at the same time, the system first obtains the columns specified by columnsToGet and then filters the returned columns.
maxVersions	The maximum number of read versions. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data within a range or a version of data is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
timeRange	Reads data within a range of versions or a version of data. For more information, see TimeRange. **Note** You must set at least one of the following parameters: maxVersions and timeRange. ◦ If you specify only maxVersions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only timeRange, all data within a range or a version of data is returned. ◦ If you specify both maxVersions and timeRange, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ To query data within a range, you must set startTime and endTime. StartTime specifies the start timestamp. EndTime specifies the end timestamp. The time range is a left-closed and right-open interval, which is [start_time, end_time). ◦ To query data within a range of versions, set SpecificTime. SpecificTime specifies a specific timestamp. Only one of SpecificTime and [StartTime, EndTime) is required. Valid values: [0, Int64.MaxValue). Unit: milliseconds.
filter	Filters read results on the server to return rows that meet the conditions in the filter. For more information, see Configure a filter. **Note** If columnsToGet and filter are used at the same time, the system first obtains the columns specified by columnsToGet and then filters the returned columns.

Parameter	Description
nextStartPrimaryKey	The start primary key for the next reading. The value of nextStartPrimaryKey can be used to determine whether all data is read. ◦ If the value of nextStartPrimaryKey is not empty in the response, the nextStartPrimaryKey value can be used as the start primary key for the next GetRange operation. ◦ If the value of nextStartPrimaryKey is empty in the response, all data within the range is returned.

- Examples

  The following code provides an example on how to read data within a 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
{
    // Construct a request object to read data within a specified range.
    var request = new GetRangeRequest(TableName, GetRangeDirection.Forward,
                    inclusiveStartPrimaryKey, exclusiveEndPrimaryKey);
    var response = otsClient.GetRange(request);
    // Continue the read operation if only part of data is returned.
    var rows = response.RowDataList;
    var nextStartPrimaryKey = response.NextPrimaryKey;
    while (nextStartPrimaryKey != null)
    {
        request = new GetRangeRequest(TableName, GetRangeDirection.Forward,
                        nextStartPrimaryKey, exclusiveEndPrimaryKey);
        response = otsClient.GetRange(request);
        nextStartPrimaryKey = response.NextPrimaryKey;
        foreach (RowDataFromGetRange row in response.RowDataList)
        {
            rows.Add(row);
        }
    }
    // Export data in rows. The code for this operation is ignored. For more information,
see the link of GitHub in the sample code.
    // If the task succeeds, no exception is returned.
    Console.WriteLine("Get range succeeded");
}
catch (Exception ex)
{
    // If the task fails, an exception is returned. Handle the exception.
    Console.WriteLine("Get range failed, exception:{0}", ex.Message);
}
```

For the detailed sample code, visit GetRange@GitHub.

GetRangeIterator

You can call this operation to read data by iteration.

- Operations

```
/// <summary>
/// Obtain data from multiple rows within the specified range. Return the iterator used t
o process each row of data.
/// </summary>
/// <param name="request"><see cref="GetIteratorRequest"/></param>
/// <returns>Return the <see cref="RowDataFromGetRange"/> iterator. </returns>
public IEnumerable<RowDataFromGetRange> GetRangeIterator(GetIteratorRequest request);
```

- Examples

// Read all rows in the range from (0, "a") to (1000, "xyz").

```
// 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 number of CUs consumed by iteration.
var cu = new CapacityUnit(0, 0);
try
{
    // Construct a GetIteratorRequest. Filter conditions are supported.
    var request = new GetIteratorRequest(TableName, GetRangeDirection.Forward, inclusiveS
tartPrimaryKey,
                                                exclusiveEndPrimaryKey, cu);
    var iterator = otsClient.GetRangeIterator(request);
    // Use the iterator that reads data in a traversal method.
    foreach (var row in iterator)
    {
        // Execute the processing logic.
    }
    Console.WriteLine("Iterate row succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("Iterate row failed, exception:{0}", ex.Message);
}
```

For the detailed code, visit GetRangeIterator@GitHub.

6.7. Search Index

6.7.1. Create search indexes

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A data table whose TimeToLive is set to -1 and MaxVersions is set to 1 is created.

Parameter

When you create a search index, you must specify TableName, IndexName, and IndexSchema. You must specify FieldSchemas, IndexSetting, and IndexSort in IndexSchema. The following table describes the parameters.

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.

Parameter	Description
FieldSchemas	The list of field schemas. You can configure the following parameters for each field schema: • FieldName: required. This parameter specifies the name of the field in the search index. The value is used as the column name. Type: String. A field in a search index can be a primary key column or an attribute column. • FieldType: required. This parameter specifies the type of the field. Use FieldType.XXX to set the type. For more information, see Data types of column values. • Array: optional. This parameter specifies whether the value is an array. Type: Boolean. If you set this parameter to true, the column stores data as an array. Data written to the column must be a JSON array. Example: ["a","b","c"]. Nested values are an array. If you set FieldType to Nested, skip this parameter. • Index: optional. This parameter specifies whether to create an index for the column. Type: Boolean. By default, this parameter is set to true, and Tablestore creates an inverted index or spatial index for the column. If this parameter is set to false, Tablestore does not create indexes for the column. • Analyzer: optional. This parameter specifies the type of analyzer to use. If FieldType is set to Text, you can set this parameter. Otherwise, the type of the analyzer is set to single-word tokenization. For more information about tokenization, see Tokenization. • EnableSortAndAgg: optional. This parameter specifies whether to enable the sorting and aggregation features. Type: Boolean. Before you enable the sorting feature, you need to set EnableSortAndAgg to true for a field. For more information about sorting, see Sorting and pagination. Notice • Store: optional. This parameter specifies whether to store the value of the field in the search index. Type: Boolean. If you set the Store parameter to true, you can read the value of the field directly from the search index without querying the data table. This improves query performance.
IndexSetting	The settings of the search index, including RoutingFields. RoutingFields: optional. This parameter specifies custom routing fields. You can specify some primary key columns as routing fields. Tablestore 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.

Parameter	Description
IndexSort	The presorting settings of the search index, including Sorters. If IndexSort is left empty, data is sorted by primary key. **Note** You can skip the presorting settings for the search indexes that contain the Nested field type. Sorters: required. This parameter specifies the presorting method for the search index. PrimaryKeySort and FieldSort are supported. For more information about sorting, see Sorting and pagination. • PrimaryKeySort: Data is sorted by primary key. You can configure the following parameter for PrimaryKeySort: Order: the sort order. Data can be sorted in ascending or descending order. Default value: DataModel.Search.Sort.SortOrder.ASC. • FieldSort: Data is sorted by field value. You can configure the following parameters for FieldSort: You can presort field values only when a search index is created and the sorting and aggregation features are enabled for fields in the search index. ◦ FieldName: the name of the field for sorting. ◦ Order: the sort order. Data can be sorted in ascending or descending order. Default value: DataModel.Search.Sort.SortOrder.ASC. ◦ Mode: the sorting method used when the field has multiple values.

Examples

- Example 1

  The following code provides an example on how to create a search index.

```
/// <summary>
/// Create a search index that contains the Keyword_type_col, Long_type_col, and Text_typ
e_col attribute columns. Set the type of data in Keyword_type_col to Keyword, in Long_typ
e_col to Long, and in Text_type_col to Text.
/// </summary>
/// <param name="otsClient"></param>
public static void CreateSearchIndex(OTSClient otsClient)
{
    // Set the names of the table and search index.
    CreateSearchIndexRequest request = new CreateSearchIndexRequest(TableName, IndexName)
;
    List<FieldSchema> FieldSchemas = new List<FieldSchema>() {
        new FieldSchema(Keyword_type_col,FieldType.KEYWORD){ // Set the name and type of
the field.
            index =true, // Enable the indexing feature.
            EnableSortAndAgg = true // Enable the sorting and aggregation features.
        },
        new FieldSchema(Long_type_col,FieldType.LONG){ index=true,EnableSortAndAgg=true},
        new FieldSchema(Text_type_col,FieldType.TEXT){ index=true}
    };
    request.IndexSchame = new IndexSchema()
    {
        FieldSchemas = FieldSchemas
    };
    // Call a client to create the search index.
    CreateSearchIndexResponse response = otsClient.CreateSearchIndex(request);
    Console.WriteLine("Searchindex is created: " + IndexName);
}
```

- Example 2

  The following code provides an example on how to specify IndexSort when you create a search index.

```
/// <summary>
/// Create a search index that contains the Keyword_type_col, Long_type_col, and Text_typ
e_col attribute columns. Set the type of data in Keyword_type_col to Keyword, in Long_typ
e_col to Long, and in Text_type_col to Text.
/// </summary>
/// <param name="otsClient"></param>
public static void CreateSearchIndexWithIndexSort(OTSClient otsClient)
{
    // Set the names of the table and search index.
    CreateSearchIndexRequest request = new CreateSearchIndexRequest(TableName, IndexName)
;
    List<FieldSchema> FieldSchemas = new List<FieldSchema>() {
        new FieldSchema(Keyword_type_col,FieldType.KEYWORD){ // Set the name and type of
the field.
            index = true, // Enable the indexing feature.
            EnableSortAndAgg = true // Enable the sorting and aggregation features.
        },
        new FieldSchema(Long_type_col,FieldType.LONG){ index=true,EnableSortAndAgg=true},
        new FieldSchema(Text_type_col,FieldType.TEXT){ index=true}
    };
    request.IndexSchame = new IndexSchema()
    {
        FieldSchemas = FieldSchemas,
        // Presort data by the Long_type_col column. You must index the Long_type_col col
umn and enable the sorting and aggregation features.
        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);
}
```

6.7.2. 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.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.

Examples

```
/// <summary>
/// Query the description of the search index.
/// </summary>
/// <param name="otsClient"></param>
public static void DescribeSearchIndex(OTSClient otsClient)
{
    // Set the names of the table and search index.
    DescribeSearchIndexRequest request = new DescribeSearchIndexRequest(TableName, IndexNam
e);
    DescribeSearchIndexResponse response = otsClient.DescribeSearchIndex(request);
    string serializedObjectString = JsonConvert.SerializeObject(response);
    Console.WriteLine(serializedObjectString); // Display the details of the response.
}
```

6.7.3. List search indexes

You can call the ListSearchIndex operation to obtain indexes created in the current instance or associated with a table.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

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

Examples

```
/// <summary>
/// List search index names.
/// </summary>
/// <param name="otsClient"></param>
public static void ListSearchIndex(OTSClient otsClient)
{
    // Set the name of the table.
    ListSearchIndexRequest request = new ListSearchIndexRequest(TableName);
    ListSearchIndexResponse response = otsClient.ListSearchIndex(request);
    // Obtain all search indexes associated with the table.
    foreach (var index in response.IndexInfos)
    {
        Console.WriteLine("indexname:" + index.IndexName);
        Console.WriteLine("tablename:" + index.TableName);
    }
}
```

6.7.4. Delete search indexes

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.

Examples

```
/// <summary>
/// Delete the search index.
/// </summary>
/// <param name="otsClient"></param>
public static void DeleteSearchIndex(OTSClient otsClient)
{
    // Set the names of the table and search index.
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest(TableName, IndexName);
    // Call the otsClient to delete the search index.
    DeleteSearchIndexResponse response = otsClient.DeleteSearchIndex(request);
}
```

6.7.5. Term query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
FieldName	The name of the field used to match the query conditions.
Term	The keyword used for the query to match the column values. The keyword is not tokenized but used as an exact word to match the column values. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that can match the keyword. For example, if the value of a TEXT field in a row is "tablestore is cool", the value can be tokenized into "tablestore", "is", and "cool". The row can meet query conditions when you set "tablestore", "is", or "cool" to the keyword to match the values of the TEXT field.
GetTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
Query	The query type. To use term query, set this parameter to TermQuery.
TableName	The name of the table.
IndexName	The name of the search index.
ColumnsToGet	Specifies whether to return all columns of each matched row. You can configure ReturnAll and Columns for this parameter. By default, ReturnAll is set to false, which indicates that not all columns are returned. If ReturnAll is set to false, you can configure Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/// <summary>
/// Search the table for rows where the value of Keyword_type_col exactly matches "SearchIn
dex".
/// </summary>
/// <param name="otsClient"></param>
public static void TermQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    // Specify that the total number of matched rows is returned.
    searchQuery.GetTotalCount = true;
    // Set the query type to TermQuery, the field used to match the query conditions to Key
word_type_col, and the keyword used for the query to match the field values to "SearchIndex
".
    searchQuery.Query = new TermQuery(Keyword_type_col, new ColumnValue("SearchIndex"));
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true // Specify that all columns of the matched rows are returned.
    };
    var response = otsClient.Search(request);
    // Check the value of NextToken.
```

6.7.6. Match all query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
Query	The query type. To use match all query, set this parameter to MatchAllQuery.
TableName	The name of the table.
IndexName	The name of the search index.
Limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.

Parameter	Description
GetTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
ColumnsToGet	Specifies whether to return all columns of each matched row. By default, ReturnAll is set to false, which indicates that not all columns are returned. If ReturnAll is set to false, you can configure Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/// <summary>
/// Query all the rows in the table and return the total number of the rows.
/// </summary>
/// <param name="otsClient"></param>
public static void MatchAllQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    searchQuery.Query = new MatchAllQuery();
    searchQuery.GetTotalCount = true; // Set GetTotalCount to true to return the total numb
er of rows that meet the query conditions.
    /*
    * In the result of a match all query, the value of TotalCount indicates the total numbe
r of rows in the table.
    * To query only the number of matched rows but not detailed data, you can set limit to
0. In this case, Tablestore returns the number of matched rows without data from the table.
    */
    searchQuery.Limit = 0;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    var response = otsClient.Search(request);
    // Check whether Tablestore returns all the matched data. When the value of isAllSucces
s is false, Tablestore may fail to query a part of partitions and return only a part of dat
a.
    Console.WriteLine("IsAllSuccess:" + response.IsAllSuccess);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

6.7.7. Match query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
FieldName	The name of the column used to match the query conditions. Match query applies to TEXT columns.
Text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, you can specify a TEXT column for a query, and set the analyzer to perform single-word tokenization. If you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
Query	The query type. To use match query, set this parameter to MatchQuery.
Operator	The logical operator. By default, OR is used as the logical operator, which indicates that a row matches the query conditions when one of the analyzed keywords is matched. If you set operator to AND, a row meets the query condition only when all tokenized keywords are matched.

Parameter	Description
MinimumShouldMatch	The minimum number of matched tokenized keywords contained in a column value. A row is returned only when the value of the queried column in the row contains at least the minimum number of matched tokenized keywords. **Note** minimumShouldMatch must be used together with the OR logical operator.
TableName	The name of the table.
IndexName	The name of the search index.
GetTotalCount	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
ColumnsToGet	Specifies whether to return all columns of each matched rows. By default, ReturnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/// <summary>
/// Use fuzzy matching and phase or approximate query to return the specified columns.
/// </summary>
/// <param name="otsClient"></param>
public static void MatchQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    // Set the query type to MatchQuery, the column used to match the query conditions to T
ext_type_col, and the keyword used for the query to match the column values 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);
}
```

6.7.8. Phrase query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
FieldName	The name of the column used to match the query conditions. Match query applies to TEXT columns.

Parameter	Description
Text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore" and "this is a table". Query results such as "this table is ..." or "is this a table" cannot be obtained.
Query	The query type. To use phrase query, set this parameter to MatchPhraseQuery.
TableName	The name of the table.
IndexName	The name of the search index.
GetTotalCount	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
ColumnsToGet	Specifies whether to return all columns of each matched rows. By default, ReturnAll is set to false, which indicates that not all columns are returned. If returnAll is set to false, you can use Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/// <summary>
/// Phrase query is similar to match query. However, a keyword is matched in a match query,
whereas a phase is matched in a phase query.
/// </summary>
/// <param name="otsClient"></param>
public static void MatchPhraseQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    // Set the query type to MatchPhraseQuery.
    searchQuery.Query = new MatchPhraseQuery(Text_type_col, "TableStore 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);
}
```

6.7.9. Prefix query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
FieldName	The name of the field used to match the query conditions.
Prefix	The value of the prefix. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that contains the specified prefix.
GetTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
Query	The query type. To use prefix query, set this parameter to PrefixQuery.

Parameter	Description
TableName	The name of the table.
IndexName	The name of the search index.
ColumnsToGet	Specifies whether to return all columns of each matched row. You can configure ReturnAll and Columns for this parameter. By default, ReturnAll is set to false, which indicates that not all columns are returned. If ReturnAll is set to false, you can configure Columns to specify the columns to return. If you do not specify the columns to return, only the primary key columns are returned. If ReturnAll is set to true, all columns are returned.

Examples

```
/// <summary>
/// Perform prefix query.
/// </summary>
/// <param name="otsClient"></param>
public static void PrefixQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    // Set the query type to PrefixQuery, the field used to match the query conditions to K
eyword_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 // Specify that all columns of the matched rows are returned.
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

6.7.10. Range query

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description

Parameter	Description
GetTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
Query	The query type. To use range query, set this parameter to RangeQuery.
FieldName	The name of the field used to match the query conditions.
From	The value from which the query starts.
To	The value with which the query ends.
IncludeLower	Specifies whether to include the value of the From parameter in the response. The value type is Boolean.
IncludeUpper	Specifies whether to include the value of the To parameter in the response. The value type is Boolean.
TableName	The name of the table.
IndexName	The name of the search index.

Examples

```
/// <summary>
/// Perform a range query. Specify a range that starts from the From value and ends at the
To value. Search for data 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));
    // Specify that the value of the From parameter is included in the response. The rows w
here the values of the specified query column are greater than or equal to 0 is returned.
    rangeQuery.IncludeLower = true;
    searchQuery.Query = rangeQuery;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

6.7.11. Wildcard query

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

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
Query	The type of the query. Set the query type to WildcardQuery.
FieldName	The name of the column.
Value	The string that contains wildcards. The string cannot exceed 32 characters in length.
TableName	The name of the data table.
IndexName	The name of the search index.
GetTotalCount	Specifies whether to return the total number of rows that match the query conditions. By default, the value of this parameter is false, which specifies that the total number of rows that match the query conditions is not returned. If you set the GetTotalCount parameter to true, the query performance is compromised.
ColumnsToGet	Specifies whether to return all columns of each matched row. By default, the value of ReturnAll is false, which specifies that not all columns are returned. If ReturnAll is set to false, you can use ColumnsToGet to specify the columns that you want to return. If you do not specify the columns that you want to return, only the primary key columns are returned. If you set the ReturnAll parameter to true, all columns are returned.

Examples

```
/// <summary>
/// Wildcard query. You can use an asterisk (*) and a question mark (?) as wildcards for wi
ldcard queries. An asterisk (*) matches zero or more characters and a question mark (?) mat
ches exactly one character.
/// </summary>
/// <param name="otsClient"></param>
public static void WildcardQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    // Set the query type to WildcardQuery. The string based on which you want to match col
umn values 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);
}
```

6.7.12. Geo query

Geo query includes the following query types: geo-distance query (GeoDistanceQuery), geo-bounding box query (GeoBoundingBoxQuery), and geo-polygon query (GeoPolygonQuery).

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Geo-distance query

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

- Parameters

Parameter	Description
FieldName	The column name. Type: Geopoint.
CenterPoint	The coordinate pair of the central point. The coordinate pair consists of latitude and longitude values. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of the longitude: [-180,180]. Example: "35.8,-45.91".

Parameter	Description
DistanceInMeter	The radius of the circle centered on the specified location. Type: Double. Unit: meters.
Query	The query statement for the search index. Sets the query type to GeoDistanceQuery.
TableName	The name of the table.
IndexName	The name of the search index.

- Examples

```
/// <summary>
/// Search the table for rows where the value of Col_GeoPoint falls within a specified di
stance from a specified central point.
/// </summary>
/// <param name="client"></param>
public static void GeoDistanceQuery(OTSClient client)
{
    SearchQuery searchQuery = new SearchQuery();
    GeoDistanceQuery geoDistanceQuery = new GeoDistanceQuery();  // Set the query type to
GeoDistanceQuery.
    geoDistanceQuery.FieldName = Geo_type_col;
    geoDistanceQuery.CenterPoint = "10,11"; // Specify the coordinate pair for a central
point.
    geoDistanceQuery.DistanceInMeter = 10000; // Set the distance from the central point
to a value greater than or equal to 10,000. Unit: meters.
    searchQuery.Query = geoDistanceQuery;
    SearchRequest searchRequest = new SearchRequest(TableName, IndexName, searchQuery);
    ColumnsToGet columnsToGet = new ColumnsToGet();
    columnsToGet.Columns = new List<string>() { Geo_type_col };  // Specify that the Col_
GeoPoint column is returned.
    searchRequest.ColumnsToGet = columnsToGet;
    SearchResponse response = client.Search(searchRequest);
    Console.WriteLine(response.TotalCount);
}
```

Geo-bounding box query

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

- Parameters

Parameter	Description
FieldName	The column name. Type: Geopoint.
TopLeft	The coordinate pair of the upper-left corner of the rectangular geographic area.

Parameter	Description
BottomRight	The coordinate pair of the lower-right corner of the rectangular geographic area. This parameter value must be in the format of "latitude,longitude". Valid values of the latitude: [-90,90]. Valid values of the longitude: [-180,180]. Example: "35.8,-45.91".
Query	The query statement for the search index. Sets the query type to GeoBoundingBoxQuery.
TableName	The name of the table.
IndexName	The name of the search index.

- Examples

```
/// <summary>
/// The data type of geo_type_col is GeoPoint. The coordinate pair of the upper-left corn
er is "10,0". The coordinate pair of the lower-right corner is "0,10". You can obtain the
rows where the value of geo_type_col falls within the rectangular geographic area.
/// </summary>
/// <param name="client"></param>
public static void GeoBoundingBoxQuery(OTSClient client)
{
    SearchQuery searchQuery = new SearchQuery();
    GeoBoundingBoxQuery geoBoundingBoxQuery = new GeoBoundingBoxQuery(); // Set the query
type to GeoBoundingBoxQuery.
    geoBoundingBoxQuery.FieldName = Geo_type_col; // Set the column name.
    geoBoundingBoxQuery.setTopLeft("10,0"); // Specify the coordinate pair for the upper-
left corner of the rectangular geographic area.
    geoBoundingBoxQuery.setBottomRight("0,10"); // Specify coordinate pair for the lower-
right corner of the rectangular geographic area.
    searchQuery.Query = geoBoundingBoxQuery;
    SearchRequest searchRequest = new SearchRequest(TableName, IndexName, searchQuery);
    var columnsToGet = new ColumnsToGet();
    columnsToGet.Columns = new List<string> { Geo_type_col };  // Specify that the Col_Ge
oPoint column is returned.
    searchRequest.ColumnsToGet = columnsToGet;
    SearchResponse response = client.Search(searchRequest);
    Console.WriteLine(response.TotalCount);
}
```

Geo-polygon query

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

- Parameters

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

- Examples

```
/// <summary>
/// Query the table for rows where the value of geo_type_col falls within a specified geo
graphic polygon area.
/// </summary>
/// <param name="client"></param>
public static void GeoPolygonQuery(OTSClient client)
{
    SearchQuery searchQuery = new SearchQuery();
    GeoPolygonQuery geoPolygonQuery = new GeoPolygonQuery();  // Set the query type to Ge
oPolygonQuery.
    geoPolygonQuery.FieldName = Geo_type_col;
    geoPolygonQuery.Points = new List<string>() { "0,0", "10,0", "10,10" }; // Specify co
ordinate pairs for vertices of a geographic polygon area.
    searchQuery.Query = geoPolygonQuery;
    SearchRequest searchRequest = new SearchRequest(TableName, IndexName, searchQuery);
    ColumnsToGet columnsToGet = new ColumnsToGet();
    columnsToGet.Columns = new List<string>() { Geo_type_col };  // Specify that the Col_
GeoPoint column is returned.
    searchRequest.ColumnsToGet = columnsToGet;
    SearchResponse response = client.Search(searchRequest);
    Console.WriteLine(response.TotalCount);
}
```

6.7.13. Boolean query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
MustQueries	The list of subqueries that the query results must match. This parameter is equivalent to the AND operator.
MustNotQueries	The list of subqueries that the query results must not match. This parameter is equivalent to the NOT operator.
FilterQueries	The list of subqueries. Only rows that match all subfilters are returned. filter is similar to query except that filter does not calculate the relevance score based on the number of subfilters that the row matches.
ShouldQueries	The list of subqueries that the query results can or cannot match. This parameter is equivalent to the OR operator. Only rows that meet the minimum number of subquery conditions specified by ShouldQueries are returned. A higher overall relevance score indicates that more subquery conditions specified by ShouldQueries are met.
MinimumShouldMatch	The minimum number of subquery conditions specified by ShouldQueries that the rows must meet. If no other subquery conditions except the subquery conditions that are specified by ShouldQueries are specified, the default value of the MinimumShouldMatch parameter is 1. If other subquery conditions, such as subquery conditions specified by MustQueries, MustNotQueries, and FilterQueries, are specified, the default value of the MinimumShouldMatch parameter is 0.
TableName	The name of the data table.
IndexName	The name of the search index.

Examples

```
/// <summary>
/// A Boolean query contains one or more subqueries. Each subquery has its own type.
///MustQueries: The rows must match the conditions.
///ShouldQueries: More than one condition is included in ShouldQueries. The rows must match
at least one condition.
///MustNotQueries: The rows must not match the conditions.
///MinimumShouldMatch: The rows must meet the minimum number of conditions.
/// </summary>
/// <param name="otsClient"></param>
public static void BoolQuery(OTSClient otsClient)
{
    Console.WriteLine("\n Start bool query...");
    var searchQuery = new SearchQuery();
    searchQuery.GetTotalCount = true;
    var boolQuery = new BoolQuery();
    var shouldQuerys = new List<IQuery>();
    shouldQuerys.Add(new TermQuery(Keyword_type_col, new ColumnValue("SearchIndex")));
    shouldQuerys.Add(new TermQuery(Keyword_type_col, new ColumnValue("TableStore")));
    boolQuery.ShouldQueries = shouldQuerys;
    boolQuery.MinimumShouldMatch = 1;
    searchQuery.Query = boolQuery;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
    foreach (var row in response.Rows)
    {
        PrintRow(row);
    }
}
```

6.7.14. Nested query

You can use NestedQuery to query the data in the child rows of a nested field. You cannot directly query NESTED columns. To query a NESTED column, you must encapsulate NestedQuery by specifing the path of the NESTED field and a subquery. The subquery can be a query of any type.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description

Parameter	Description
Path	The path of the NESTED column. The path is similar to the tree structure. For example, news.title indicates the title subcolumn in the news column of the NESTED type.
Query	The query on the subcolumn in the NESTED column. The query can be of any query type.
ScoreMode	The value on which a score is calculated when multiple values exist for a column.
TableName	The name of the table.
IndexName	The name of the search index.

Examples

```
/// <summary>
/// The col1_nested column contains nested_1 and nested_2. The type is NESTED. You need to
search the col1_nested.nested_1 column for data that matches "tablestore".
/// </summary>
/// <param name="otsClient"></param>
public static void NestedQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    searchQuery.GetTotalCount = true;
    var nestedQuery = new NestedQuery();
    nestedQuery.Path = "col1_nested"; // Set the path of the NESTED field.
    TermQuery termQuery = new TermQuery("col1 _nested.nested_1",new ColumnValue("tablestore
"));// Construct a subquery of NestedQuery.
    nestedQuery.Query = termQuery;
    nestedQuery.ScoreMode = ScoreMode.None;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

6.7.15. Sorting and pagination

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

Index presorting

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

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

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

Specify a sorting method

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

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

- ScoreSort

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

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

  ```
  var searchQuery = new SearchQuery();
  searchQuery.Sort = new Sort(new List<ISorter>() { new ScoreSort() });
  ```

- PrimaryKeySort

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

  ```
  // Sort the query result based on the primary key columns in ascending order.
  var searchQuery = new SearchQuery();
  searchQuery.Sort = new Sort(new List<ISorter>() { new PrimaryKeySort() });
  // Sort the query result based on the primary key columns in descending order.
  var searchQuery = new SearchQuery();
  searchQuery.Sort = new Sort(new List<ISorter>() { new PrimaryKeySort(SortOrder.DESC) });
  ```

- FieldSort

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

  ```
  var searchQuery = new SearchQuery();
  var fieldSort = new FieldSort("col", SortOrder.ASC);
  searchQuery.Sort = new Sort(new List<ISorter>() { fieldSort });
  ```

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

```
var searchQuery = new SearchQuery();
var col1Sort = new FieldSort("col", SortOrder.ASC);
var col2Sort = new FieldSort("co2", SortOrder.ASC);
searchQuery.Sort = new Sort(new List<ISorter>() { col1Sort, col2Sort });
```

- GeoDistanceSort

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

```
var searchQuery = new SearchQuery();
var geoDistanceSort = new GeoDistanceSort("geoCol", new List<string>(){"0","0"});
searchQuery.Sort = new Sort(new List<ISorter>() { geoDistanceSort });
```

Specify a pagination method

You can use Limit and Offset or Token for pagination when you obtain returned results.

- Use the Limit and Offset parameters

  When the total number of rows to obtain is smaller than 50,000, you can specify the Limit and Offset parameters for pagination. The sum of the Limit and Offset parameter values cannot exceed 50,000. The maximum value of Limit is 100.

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

  If you use Limit and Offset for pagination but do not configure the Limit and Offset values, the default values are used. The default value of Limit is 10. The default value of Offset is 0.

```
var searchQuery = new SearchQuery();
searchQuery.Query = new MatchAllQuery();
searchQuery.Limit = 100;
searchQuery.Offset = 100;
```

- Use a token

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

  If Tablestore cannot complete reading data that meets the query condition, the server returns NextToken. You can use NextToken to continue reading the subsequent data.

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

  When you use the token, the sorting method is the same as that used in the previous request. Tablestore sorts data based on the IndexSort field by default or based on the method that you have specified. Therefore, you cannot set the sorting method if you use a token. You cannot set Offset when a token is used. Data is returned page by page, which results in a slow query.

  > **Notice** Search indexes of the NEST type do not support IndexSort. If you use a search index of the NEST type to query data and require pagination, you must specify the sorting method to return data in the query conditions. Otherwise, Tablestore does not return NextToken when all data that meets the query conditions is not completely read.

```
/// <summary>
/// If you use a token for pagination, Tablestore reads all data and returns the data in
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) // Tablestore stops reading when the value of NextT
oken is null. null indicates that Tablestore reads all data.
    {
        request.SearchQuery.Token = response.NextToken;
        response = otsClient.Search(request);
        rows.AddRange(response.Rows);
    }
    return response;
}
```

6.8. Global secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A table is created. The value of timeToLive is -1. The value of maxVersions is 1.
- Predefined columns are set for the table.

Create an index table by calling the CreateGlobalIndex operation

You can call the CreateIndex operation to create an index table for an existing base table.

> **Note** You can also create one or more index tables when you create a base table by calling the CreateTable operation. For more information, see Create tables.

- Parameters

Parameter	Description
mainTableName	The name of the table.

Parameter	Description
indexMeta	The schema information of the index table, which includes the following items: ○ IndexName: the name of the index table. ○ primaryKey: the primary key of the index table, which is a combination of primary key columns and predefined columns of the base table. ○ definedColumns: the indexed attribute column, which is a combination of predefined columns of the base table. ○ indexUpdateMode: the update mode of the index table. Only IUM_ASYNC_INDEX is supported. ○ indexType: the type of the index table. Only IT_GLOBAL_INDEX is supported.
includeBaseData	Specifies whether the index table includes the existing data in the base table. When the includeBaseData parameter is set to true, the index includes the existing data. When the value is false, the index excludes the existing data.

- Examples

```
public static void CreateGlobalIndex()
{
    OTSClient otsClient = Config.GetClient();
    Console.WriteLine("Start create globalIndex...") ;
    IndexMeta indexMeta = new IndexMeta(IndexName2);
    indexMeta.PrimaryKey = new List<string>() { Col2 };
    indexMeta.DefinedColumns = new List<string>() { Pk1 };
    CapacityUnit reservedThroughput = new CapacityUnit(0, 0);
    CreateGlobalIndexRequest request = new CreateGlobalIndexRequest(TableName, indexMeta)
;
    otsClient.CreateGlobalIndex(request);
    Console.WriteLine("Global Index is created,tableName: " + TableName + ",IndexName:" +
IndexName2);
 }
```

Read data from the index table

You can read a row of data or read data within a specified range from the index table. If the columns to return are included in the index table, you can query the data from the index table. Otherwise, you must query the base table for the required data.

- Read a row of data from the index table

  For more information, see Single-row operations.

  When you use the GetRow operation to read data from the index table, take note of the following items:

  ○ You must set tableName to the name of the index table.

- Tablestore autocompletes the primary key that is not configured in the index columns for the index table. Therefore, when you specify the primary key of a row, you must specify the indexed columns based on which you create the index table and the autocompleted primary key columns.

- Read data within a specified range from the index table

  For more information, see Multi-row operations.

  - Parameters

    When you use the GetRange operation to read data from the index table, take note of the following items:

    - You must set tableName to the name of the index table.
    - Tablestore autocompletes the primary key that is not configured in the index columns for the index table. Therefore, when you specify the start primary key and the end primary key of the range, you must specify the indexed columns based on which you create the index table and the autocompleted primary key columns.

  - Examples

```
public static void GetRangeFromIndexTable()
{
    Console.WriteLine("Start getRange from index...") ;
    OTSClient otsClient = Config.GetClient();
    // Specify the value of the first primary key column Col1 and read the data.
    PrimaryKey inclusiveStartPrimaryKey = new PrimaryKey
    {
        { Col1, new ColumnValue("Col1Value") },
        { Pk1,  ColumnValue.INF_MIN },
        { Pk2, ColumnValue.INF_MIN }
    };
    PrimaryKey exclusiveEndPrimaryKey = new PrimaryKey
    {
        { Col1, new ColumnValue("Col1Value") },
        { Pk1,  ColumnValue.INF_MAX },
        { Pk2, ColumnValue.INF_MAX }
    };
    GetRangeRequest request = new GetRangeRequest(IndexName, GetRangeDirection.Forward,
inclusiveStartPrimaryKey, exclusiveEndPrimaryKey);
    GetRangeResponse response = otsClient.GetRange(request);
    IList<Row> rows = response.RowDataList;
    PrimaryKey nextStartPrimaryKey = response.NextPrimaryKey;
    while (nextStartPrimaryKey != null)
    {
        request = new GetRangeRequest(TableName, GetRangeDirection.Forward, nextStartPr
imaryKey, exclusiveEndPrimaryKey);
        response = otsClient.GetRange(request);
        nextStartPrimaryKey = response.NextPrimaryKey;
        foreach (var row in response.RowDataList)
        {
            rows.Add(row);
        }
    }
    foreach (var row in rows)
    {
        PrintRow(row);
    }
    Console.WriteLine("TotalRowsRead: " + rows.Count);
}
private static void PrintRow(Row row)
{
    Console.WriteLine("-----------------");
    foreach (KeyValuePair<string, ColumnValue> entry in row.GetPrimaryKey())
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    foreach (Column entry in row.GetColumns())
    {
        Console.WriteLine(entry.Name + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("-----------------");
}
```

Delete the index table by calling the DeleteGlobalIndex operation

You can call the DeleteIndex operation to delete a specified index table from a base table.

- Parameters

Parameter	Description
mainTableName	The name of the table.
indexName	The name of the index table.

- Examples

```
public static void DeleteGlobalIndex()
{
    OTSClient otsClient = Config.GetClient();
    Console.WriteLine("Start delete globalIndex...") ;
    DeleteGlobalIndexRequest request = new DeleteGlobalIndexRequest(TableName, IndexName)
;
    otsClient.DeleteGlobalIndex(request);
    DeleteGlobalIndexRequest request2 = new DeleteGlobalIndexRequest(TableName, IndexName
2);
    otsClient.DeleteGlobalIndex(request2);
    Console.WriteLine("Global Index is deleted,tableName: " + TableName + ",IndexName:" +
IndexName + "," + IndexName2);
}
```

6.9. Troubleshooting

This topic describes the error handling methods in Tablestore SDK for .NET.

Methods

Tablestore SDK for Java handles errors as `exceptions`. The operation is successful if no exceptions are returned for the called operation. If an exception is returned, the operation fails.

> **Note** Batch API operations such as BatchGetRow and BatchWriteRow are called only when the status check of each row is successful.

Exceptions

Tablestore SDK for .NET has two types of exceptions: OTSClientException and OTSServerException. Both are inherited from Exception.

- OTSClientException: an internal SDK exception such as an incorrect parameter value or a failure to parse the results returned by the server.
- OTSServerException: a server error. When a server error occurs, the server parses and returns the error message to the client. OTSServerException has the following components:
  - HttpStatusCode: an HTTP status code such as 200 or 404.
  - ErrorCode: an error type string returned by Tablestore.
  - ErrorMessage: an error message string returned by Tablestore.

- RequestId: the UUID that uniquely identifies a request. If a problem persists, submit the request ID to Tablestore technical support personnel to seek help.

Retry

- Tablestore SDK for .NET automatically retries an operation when an error occurs. In the default retry policy, the maximum retry attempts is 3, and the maximum retry interval is 2s.
- You can modify RetryPolicy in OTSClientConfig to customize a retry policy.

7.PHP SDK

7.1. Preface

This topic describes how to use Tablestore SDK for PHP. This topic applies to Tablestore SDK for PHP 4.0.0 and later. This topic assumes that you have activated Tablestore and created an AccessKey ID and an AccessKey secret.

- If you have not activated Tablestore or want to learn more about Tablestore, see the Tablestore product page.
- An AccessKey ID and an AccessKey secret are created. For more information, see Obtain an AccessKey pair.

Precautions

Tablestore SDK for PHP 4.0.0 and later support max versions and time to live (TTL). However, these SDK versions are not compatible with Tablestore SDK for PHP 2.x.x. For more information, see the following topics:

- Max versions and TTL
- Max versions and TTL
- Configure an auto-increment primary key column

Download Tablestore SDK for PHP

- SDK source code package
- GitHub

For the versions of Tablestore SDK for PHP, see PHP SDK.

Versions

Latest version: 5.0.0.

Compatibility

- Tablestore SDK for PHP 3.x.x: compatible.
- Tablestore SDK for PHP 2.x.x: incompatible.

Change records

- 5.0.0
  - New features: Local transactions are supported.
  - New features: Search indexes are supported.
  - New features: Global secondary indexes are supported.
- 4.1.0

  Basic Stream-based operations are supported.
- 4.0.0
  - PHP 5.5 and later are supported, including 5.5, 5.6, 7.0, 7.1, and 7.2. Only 64-bit PHP systems are supported. We recommend that you use PHP 7.

- New features: TTL settings are supported. The table_options parameter is added to CreateTable and UpdateTable.
- New features: Max versions is supported. Timestamp settings are supported for PutRow, UpdateRow, DeleteRow, and BatchGetRow. The max_versions parameter is added to GetRow, GetRange, and BatchGet.
- New features: The auto increment feature is supported for primary key columns. The return_type parameter is added to API operations. The primary_key response parameter is added for the corresponding operation to return the primary key.
- Changes: The underlying Protobuf is upgraded to the Google protobuf-php library.
- Changes: The value of the primary_key parameter for each API 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 API operation is changed to the LIST type to support max versions.

7.2. Install Tablestore SDK for PHP

This topic describes how to install and use Tablestore SDK for PHP.

Prerequisites

- 64-bit PHP 5.5 or later is available.

  You can run the **php -v** command to view the current PHP version.

  Tablestore uses 64-bit integers. However, in 32-bit PHP, only the STRING type can be used to represent 64-bit INTEGER values. Therefore, Tablestore does not support 32-bit PHP. In Windows, PHP versions earlier than PHP 7 are not actually 64-bit. When you run PHP in Windows, you must upgrade PHP to PHP 7 or modify the environment yourself. We recommend that you use PHP 7 to obtain the optimal performance.

- cURL extension (optional)

  You can run the **php -m** command to check whether the cURL extension is installed.

  > **Note**
  > - In Ubuntu, you can run the sudo apt-get install php-curl command to install the cURL extension of PHP by using apt-get.
  > - In CentOS, you can run the sudo yum install php-curl command to install the cURL extension of PHP by using YUM.

- OpenSSL extension (optional)

  You must install the OpenSSL PHP extension if you use HTTPS.

Installation methods

- Composer

  To install Tablestore SDK for PHP by using Composer, perform the following operations:

  i. Run the `composer require aliyun/aliyun-tablestore-sdk-php` command in the root directory of the project or declare the dependency on Tablestore SDK for PHP in the composer.json file.

```
"require": {
    "aliyun/aliyun-tablestore-sdk-php": "~5.0"
}
```

ii. Run the composer install command to install the dependency. After the dependency is installed, check whether your directory structure complies with the following structure:

```
.
├── 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 import the dependency into app.php.

```
require_once __DIR__ . '/vendor/autoload.php';
```

> **Note**
>
> - If your project already has autoload.php imported, you do not need to import autoload.php again after you add the SDK dependency.
> - If a network error occurs when you use Composer, you can enter the following command on the command line to use Packagist that is available to China: `composer config -g repo.packagist composer https://developer.aliyun.com/composer`. For more information about Packagist, see Images.

- Source code package

  You can download the source code package by using the following methods:

  - Select the required version and download the ZIP package at GitHub.
  - Click SDK source code to download the source code package.

Sample programs

Tablestore SDK for PHP provides a variety of sample programs for your reference or use.

You can obtain the sample programs by using one of the following methods:

- Download and decompress Tablestore SDK for PHP. Find the sample programs in the examples directory.
- Access the GitHub project for Tablestore SDK for PHP at aliyun-tablestore-php-sdk.

To run a sample program, perform the following operations:

1. Decompress the downloaded SDK package.
2. Modify the ExampleConfig.php file in the examples directory.

```
EXAMPLE_END_POINT: the domain name that you use to access the region where your Tablest
ore instance is located. Example: https://sun.cn-hangzhou.ots.aliyuncs.com.
EXAMPLE_ACCESS_KEY_ID: the AccessKey ID that you obtained from Alibaba Cloud.
EXAMPLE_ACCESS_KEY_ID: the AccessKey secret that you obtained from Alibaba Cloud.
EXAMPLE_INSTANCE_NAME: the instance that you use to run the sample program. The sample
program is operated in this instance.
```

3. Run a sample file in the examples directory separately.

The following table lists the sample files.

File	Description
NewClient.php	Demonstrates how to configure the default client.
NewClient2.php	Demonstrates how to customize the client.
NewClientLogClosed.php	Demonstrates how to disable the log display of the client.
NewClientLogDefined.php	Demonstrates how to customize the log display of 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.
GetRow2.php	Demonstrates how to set column_to_get in GetRow.

File	Description
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 primary key columns.
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 specified columns for multiple rows from a table.
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 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.
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.

File	Description
GetRangeWithColumnFilter.php	Demonstrates how to use GetRange with conditional filtering.

7.3. Initialization

OTSClient is the client for Tablestore. OTSClient provides a series of methods for you to manage tables and perform read and write operations on one or more rows. To use Tablestore SDK for PHP to initiate a request, you must initialize an OTSClient instance and modify the default configuration items of OTSClientConfig.

Obtain an endpoint

An endpoint is a domain name that is used to access a Tablestore instance in a region. The following table describes the examples of endpoints in the supported formats.

Example	Description
http://sun.cn-hangzhou.ots.aliyuncs.com	The public endpoint that is used to access the sun instance in the China (Hangzhou) region over HTTP.
https://sun.cn-hangzhou.ots.aliyuncs.com	The public endpoint that is used to access the sun instance in the China (Hangzhou) region over HTTPS.

Notice You can connect to Tablestore by using public endpoints or private endpoints. For more information, see Endpoint.

To query the endpoints of your Tablestore instance, perform the following steps:

1. Log on to the Tablestore console.
2. Click the name of the instance to go to the **Instance Details** tab.

   On the Instance Details tab, you can view the endpoints of the instance in the Instance Access URL section.

Configure an AccessKey pair

To access Tablestore, you must have an AccessKey pair that consists of an AccessKey ID and AccessKey secret to verify your identity. The following types of AccessKey pairs are supported:

- The AccessKey pair of an Alibaba Cloud account. To obtain the AccessKey pair of an Alibaba Cloud account, take the following steps:
  i. On the Alibaba Cloud official website,Create Your Alibaba Cloud Account.
  ii. Create an AccessKey ID and an AccessKey secret. For more information, see Obtain an AccessKey pair.
- The AccessKey pair of a RAM user who is granted with permissions to access Tablestore. To obtain the AccessKey pair of a RAM user, take the following steps:
  i. Use your Alibaba Cloud account to log on to RAM. Create a RAM user or use an existing RAM user.
  ii. Use your Alibaba Cloud account to authorize the RAM user to access Tablestore.

iii. After the RAM user is authorized, you can use the AccessKey pair of the RAM user to access Tablestore.

- Temporary access credentials obtained from STS. To obtain temporary access credentials from STS, take the following steps:
  i. The application server uses RAM or STS to obtain access credentials that consist of a temporary AccessKey ID, an AccessKey secret, and a token. After the access credentials are obtained, the application server sends them to you.
  ii. You can use the received access credentials to access Tablestore.

Initialize an OTSClient instance

After you obtain the AccessKey ID and AccessKey secret, you can perform the following operations to initialize an OTSClient instance:

1. Use the endpoint of a Tablestore instance to create a client.

   Examples

```
$otsClient = new Aliyun\OTS\OTSClient(array(
    'EndPoint' => "<your endpoint>",
    'AccessKeyID' => "<your access id>",
    'AccessKeySecret' => "<your access key>",
    'InstanceName' => "<your instance name>"
));
```

2. Configure the OTSClient instance.

   To modify the default configurations of the OTSClient instance, import the corresponding parameters such as the proxy, connection timeout time, and the maximum number of connections when you create the OTSClient instance. The following table lists the parameters you can configure when you create the OTSClient instance.

Parameter	Description	Default Value
ConnectionTimeout	The maximum latency allowed to connect to Tablestore.	2.0 seconds
StsToken	The STS 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 large value when large volumes of data are transmitted.
RetryPolicy	The retry policy.	DefaultRetryPolicy. A value of null indicates that RetryPolicy is disabled.
ErrorLogHandler	The function to process error logs, which is used to display the logs of errors returned by Tablestore.	defaultOTSErrorLogHandler. A value of null indicates that ErrorLogHandler is disabled.

Parameter	Description	Default Value
DebugLogHandler	The function to process debug logs, which is used to display the logs of normal requests and responses.	defaultOTSDebugLogHandler. A value of null indicates that defaultOTSDebugLogHandler is disabled.

HTTPS

Install the OpenSSL PHP extension.

7.4. Table operations

7.4.1. Create tables

This topic describes how to create a data table by calling the CreateTable operation. When you call the CreateTable operation, you must specify the schema information and configuration information of the data table when you call the CreateTable operation. You can set reserved read throughput and reserved write throughput for data tables in high-performance instances. You can create one or more index tables when you create a data table.

> **Note**
>
> - After a data table is created, it takes several seconds to load the data table. During this period, read and write operations performed on the table may fail. Perform operations on the data table after the data table is loaded.
> - You must specify the primary key when you create a data table. A primary key can contain one to four primary key columns. Each primary key column has a name and a data type.

Prerequisites

- An instance is created in the console. For more information, see Create instances.
- OTSClient is initialized. For more information, see Initialization.

Operations

```
/**
 * Create a table. Set the number, names, sequence, and types of primary key columns, reser
ved read and write throughput, TTL, and Stream options.
 * @api
 * @param [] $request The request parameters.
 * @return [] The response is empty. If the CreateTable operation succeeds, no message is r
eturned. An empty array is returned to be consistent with other operations.
 * @throws OTSClientException The exception that is returned when a parameter error occurs
or the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that is returned when the Tablestore server ret
urns an error.
 */
public function createTable(array $request);
```

Parameters

- Request parameters

Parameter	Description
table_meta	The schema information of the table, which includes the following items: ○ table_name: the name of the table. ○ primary_key_schema: the schema of the primary key for the table. For more information, see Primary keys and attributes. **Note** Attribute columns do not need to be defined. Different rows in Tablestore can have different attribute columns. The names of attribute columns can be specified when data is written. ■ Each item consists of the name, type (PrimaryKeyType), and configurations (PrimaryKeyOption, which is optional) of the primary key columns. PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, or PrimaryKeyTypeConst::CONST_BINARY, which separately 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 Configure an auto-increment primary key column. ■ The primary key of a table can contain one to four primary key columns. Primary key columns are sorted in the order in which they are added. For example, PRIMARY KEY (A, B, C) and PRIMARY KEY (A, C, B) have different primary key schemas. Tablestore 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 stored in the same partition. Therefore, we recommend that you assign no more than 10 GB of data with the same partition key value. Otherwise, a single partition may be too large to split. We recommend that you set a well-designed primary key so that read and write operations can be evenly distributed to different partitions to facilitate load balancing. ○ defined_column: the predefined columns of the table and the data types of the predefined column values. Primary key columns cannot be set to predefined columns. You can use predefined columns as the indexed columns or attribute columns for index tables.
	The configurations of the table. For more information, see Data versions and TTL.

Parameter	Description
table_options	The configurations include the following items: ○ time_to_live: the period for which data in the table can be retained. This period is the validity period of data. When the retention period exceeds the time_to_live value, Tablestore automatically deletes expired data. The minimum time_to_live value is 86,400 (one day). A value of -1 indicates that data never expires. When you create a table, you can set timeToLive to -1 if you expect that data never expires. After the table is created, you can call the UpdateTable operation to modify the time_to_live value. Unit: seconds. **Note** You must set time_to_live to -1 if you want to create index tables for the base table. ○ max_versions: the maximum number of versions of data that can be retained for a single attribute column. When the number of versions of data in attribute columns exceeds the max versions value, Tablestore automatically deletes data of earlier versions. When you create a table, you can customize the maximum number of versions that can be retained for the data in an attribute column. After the table is created, you can call the UpdateTable operation to modify the max_versions value for the table. **Note** You must set max_versions to 1 if you want to create index tables for the base table. ○ deviation_cell_version_in_sec: the max version offset, which is the maximum difference between the timestamp of the written data and the current system time. The difference between the version number and the data written time must be within the value of deviation_cell_version_in_sec. Otherwise, an error occurs when the data is written. Unit: seconds. The valid version range of data in an attribute column is calculated based on the formula: Valid version range = **[Data written time - Max version offset, Data written time + Max version offset)**. When you create a table, Tablestore uses the default value of 86400 if you do not specify a max version offset. After the table is created, you can call the UpdateTable operation to modify the deviation_cell_version_in_sec value. Unit: seconds.

Parameter	Description
reserved_throughput	The reserved read and write throughput of the table. For capacity instances, the reserved read and write throughput can be set only to 0. Reserved throughput does not apply to these instances. The default value 0 indicates that all throughput is billed on a pay-as-you-go basis. Unit: CU. ○ When reserved read throughput or reserved write throughput is set to a value greater than 0, Tablestore allocates and reserves related resources for the table. After the table is created, Tablestore charges reserved throughput resources. Additional throughput is billed on a pay-as-you-go basis. For more information, see Billing overview. ○ When reserved read throughput or reserved write throughput is set to 0, Tablestore does not allocate or reserve related resources for the table.
stream_spec	Stream configurations (optional). ○ enable_stream: indicates whether Stream is enabled for the table. ○ expiration_time: the expiration time of the Stream data of the table. Unit: hour. Earlier modification records are deleted. This parameter can be set only when the enable_stream parameter is set to true.
index_metas	The schema information of the index tables. Each index_meta includes the following items: ○ name: the name of the index table. ○ primary_key: the primary key of the index table, which is a combination of primary key columns and predefined columns of the base table. ○ defined_column: the indexed attribute column, which is a combination of predefined columns of the base table. ○ index_update_mode: the update mode of the index table. Only IUM_ASYNC_INDEX is supported. ○ index_type: the type of the index table. Only IT_GLOBAL_INDEX is supported.

- Request format

```
$result = $client->createTable([
    'table_meta' => [                  // Set the schema information of the table. This p
arameter is required.
        'table_name' => '<string>',
        'primary_key_schema' => [
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>, <PrimaryKeyOption>]
        ]
    ],
    'reserved_throughput' => [         // Set the reserved read and write throughput of t
he table. This parameter is required.
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'table_options' => [              // Set the configuration information of the table.
This parameter is required.
        'time_to_live' => <integer>,
        'max_versions' => <integer>,
        'deviation_cell_version_in_sec' => <integer>
    ],
    'stream_spec' => [
        'enable_stream' => true || false,
        'expiration_time' => <integer>
    ]
]);
```

- Response parameters

  The response is empty. If an error occurs, the system returns an exception.

- Response format

```
[]
```

Examples

- Create a base table without creating an index table

  The following code provides an example on how to create a table that contains three primary key columns. Set the reserved read and write throughput to 0, TTL to -1, and max_versions to 2, and enable Stream.

```
//Create a schema for the primary key columns, which includes the number, names, and type
s of primary key columns.
//The first primary key column is named pk0 and requires an INTEGER value. The first prim
ary key column is also the partition key.
//The second primary key column is named pk1, and requires a STRING value.
// The third primary key column is named PK2 and requires a BINARY value.
$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
    ]
]);
```

- Create an index table when you create the base table

```
$request = array (
    'table_meta' => array (
        'table_name' => self::$tableName, // The name of the base table is testGlobalTabl
eName.
        'primary_key_schema' => array (
            array('PK0', PrimaryKeyTypeConst::CONST_INTEGER),
            array('PK1', PrimaryKeyTypeConst::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 throughput and reserved write thro
ughput to 0.
            'write' => 0
        )
    ),
    'table_options' => array(
        'time_to_live' => -1,   // Set time_to_live to -1, which indicates that the data
never expires. You must set time_to_live to -1 for the base table that has index tables.
        'max_versions' => 1,    // Specify the maximum number of versions that can be ret
ained for each column. A value of 1 indicates that only the latest version is retained fo
r each column. You must set max_versions to 1 for the base table that has index tables.
        'deviation_cell_version_in_sec' => 86400   // Set the max version offset of data.
Unit: seconds.
    ),
    'index_metas' => 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);
```

7.4.2. UpdateTable

This topic describes the UpdateTable operation.

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.

Operations

```
    /**
     * Update a table, including the reserved read throughput and reserved write throughput
, configuration information, and Stream configuration of the table.
     * This method can be used to increase or decrease the reserved read throughput and res
erved write throughput.
     * @api
     * @param [] $request Request parameters.
     * @return [] The response.
     * @throws OTSClientException The exception that occurs when a parameter error occurs o
r the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that occurs when the Tablestore server retu
rns 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 parameters of UpdateTable are defined and configured in the same way as those of CreateTable. However, UpdateTable does not have the table_meta parameter. 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 Tablestore billing to avoid unexpected fees. You can set the reserved read throughput and reserved write throughput of capacity instances only to 0. Throughput reservation is not allowed for these instances.
  - 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 Tablestore API supports automatic data expiration. If you do not want data to expire, set time_to_live to -1.
    - Tablestore 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 Tablestore server. For more information about the data timestamp, see related documentations.
    - After the TTL is set, the server checks whether data expires based on the data timestamp. If you specify a timestamp when you write 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 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.
    - Tablestore supports multiple versions. By default, the system generates a version number for newly written 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, Tablestore provides the deviation_cell_version_in_sec parameter to limit the permissible deviation between the data writing timestamp and the current system time. The unit of this parameter is second. You can specify this parameter when you create 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/write throughput configuration, which is 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 Tablestore billing to avoid unexpected fees. You can set the reserved read throughput and reserved write throughput of capacity instances only to 0. Throughput reservation is not allowed for these instances.
    - 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.

Examples

The following code provides an example on how to update the numbers of read CUs of a table to 1 and the numbers of write CUs to 2:

```
    $result = $client->updateTable([
        'table_name' => 'SampleTable',
        'reserved_throughput' => [
            'capacity_unit' => [
                'read' => 1,            // The number of read or write CUs can be separatel
y updated.
                'write' => 2
            ]
        ]
    ]);
```

The following code provides an example on 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' => 'SampleTable',
        'table_options' => [
            'time_to_live' => 86400,
            'max_versions' => 2,
            'deviation_cell_version_in_sec' => 10
        ]
    ]);
```

The following code provides an example on how to enable Stream for a table and set the expiration time to 24 hours:

```
    $result = $client->updateTable([
        'table_name' => 'SampleTable',
        'stream_spec' => [
            'enable_stream' => true,
            'expiration_time' => 24
        ]
    ]);
```

7.4.3. List the names of tables

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

> **Note** For more information about the ListTable operation, see ListTable.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created.

API operations

```
/**
 * Query the names of all tables created in the current instance.
 * @api
 * @param [] $request The request parameters, which are empty.
 * @return [] The response.
 * @throws OTSClientException The exception that is thrown when a parameter error occurs or
the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that is thrown when the Tablestore server retur
ns an error.
 */
public function listTable(array $request);
```

Request format

The request parameters are empty. You do not need to set request parameters.

```
$result = $client->listTable([]);
```

Result format

A STRING list is returned. Each item in the list is a table name.

```
[
    '<string>',
    '<string>',
    '<string>'
]
```

Examples

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

```
$result = $otsClient->listTable([]);
```

7.4.4. Split data by a specified size

You can call the ComputeSplitsBySize operation to logically splits the data in a table into multiple shards whose sizes are close to a specified value. The locations of the split points among these splits and the hosts where the shards reside are returned. This operation is typically used to implement plans on compute engines, such as concurrency plans.

> **Note** For more information about the ComputeSplitsBySize operation, see ComputeSplitPointsBySize.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Operations

```
    /**
     * 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 [] Response.
     * @throws OTSClientException The exception that is returned when a parameter error occurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore server returns an error.
     */
    public function computeSplitPointsBySize(array $request)
```

Parameters

- Request parameters

Parameter	Description
table_name	The name of the table.
split_size	The approximate size of each partition. Unit: 100 MB.

- Request format

```
$result = $client->ComputeSplitsBySize([
    'table_name' => '<string>', // Specify the name of the table. This parameter is required.
    'Split_size ' =>  // Set the part size. This parameter is required.
]);
```

- Response parameters

Parameter	Description

Parameter	Description
consumed	The value of capacity units consumed by this operation. capacity_unit indicates the number of read and write CUs consumed. ◦ read: the reserved read throughput. ◦ write: the reserved write throughput.
primary_key_schema	The schema of the primary key for the table, which is the same as that specified when the table is created.
splits	The split points between partitions, which includes the following items: ◦ lower_bound: the minimum value during the range of the primary key. The lower_bound value can be passed to GetRange to range read data. ■ Each item contains the primary key name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType) in sequence. ■ PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, PrimaryKeyTypeConst::CONST_BINARY, PrimaryKeyTypeConst::CONST_INF_MIN, or PrimaryKeyTypeConst::CONST_INF_MAX, which separately indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, INF_MIN(-inf), and INF_MAX(inf) types. ◦ upper_bound: the maximum value during the range of the primary key. The format of upper_bound is same as that of lower_bound. The upper_bound value can be passed to GetRange to read data by range. ◦ location: the machine where the split point is located. The value of this parameter can be empty.

- Response format

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key_schema' => [
        ['<string>', <PrimaryKeyType>],
        ['<string>', <PrimaryKeyType>, <PrimaryKeyOption>]
    ]
    'splits' => [
        [
            'lower_bound' => [
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'upper_bound' => [
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'location' => '<string>'
        ],
        // ...
    ]
]
```

Examples

The following code provides an example on how to logically splits the data in a table into multiple partitions whose sizes are close to 100 MB:

```
$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']);
}
```

7.4.5. Query the description of a table

> **Note** For more information about the DescribeTable operation, see DescribeTable.

Prerequisites

- A Tablestore client is initialized. For more information, see Initialization.
- A data table is created.

API operations

```
/**
 * Query the schema information, configuration information, reserved read throughput and re
served write throughput, and Stream configuration information about the specified table.
 * @api
 * @param [] $request The request parameters.
 * @return [] The response.
 * @throws OTSClientException The exception that is thrown when a parameter error occurs or
the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that is thrown when the Tablestore server retur
ns an error.
 */
public function describeTable(array $request);
```

Parameters

- Request parameters

Parameter	Description
table_name	The name of the table.

- Request format

```
$result = $client->describeTable([
     'table_name' => '<string>', // Specify the name of the table. The table name must be
specified.
]);
```

- Response parameters

Parameter	Description
table_meta	The schema information about the table, which includes the following items: ○ table_name: the name of the table. ○ primary_key_schema: the primary key of the table, which is the same as the primary key defined when the table is created.

Parameter	Description
capacity_unit_details	The reserved read throughput and reserved write throughput of the table, which includes the following items: ◦ capacity_unit indicates the reserved read throughput and reserved write throughput configurations of the table, which are related to billing. ▪ 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 last increased 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 last decreased 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 configuration information about the table, which includes the following items: ◦ time_to_live: the period during which data in the table can be retained. This period is the validity period of data. ◦ max_versions: the maximum number of versions of data that can be retained for a single attribute column. ◦ deviation_cell_version_in_sec: the max version offset, which is the maximum difference between the current system time and the timestamp of the written data.
stream_details	The Stream information about the table, which includes the following items: ◦ 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.

- Result format

```
[
    'table_meta' => [
        'table_name' => '<string>',
        'primary_key_schema' => [
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>, <PrimaryKeyOption>]
        ]
    ],
    '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>
    ]
]
```

Examples

The following code provides an example on how to query the description of a table:

```
$result = $client->describeTable([
        'table_name' => 'mySampleTable',
        ]);
        var_dump($result);
```

7.4.6. Delete tables

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

> **Note** For more information about the DeleteTable operation, see DeleteTable.

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A table is created.
- The index tables and search indexes for the table are deleted.

Operations

```
/**
 * Delete a table based on the table name.
 * @api
 * @param [] $request The request parameters.
 * @return [] The response is empty. If the DeleteTable operation succeeds, no message is r
eturned. An empty array is returned to be consistent with other operations.
 * @throws OTSClientException The exception that is returned when a parameter error occurs
or the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that is returned when the Tablestore server ret
urns an error.
 */
 public function deleteTable(array $request);
```

Parameters

- Request parameters

Parameter	Description
table_name	The name of the base table.

- Request format

```
$result = $client->deleteTable([
     'table_name' => '<string>', // REQUIRED
]);
```

- Response parameters

  The response is empty. If an error occurs, the system returns an exception.

- Response format

```
    []
```

Examples

The following code provides an example on how to delete a specified table:

```
$result = $otsClient->deleteTable([
    'table_name' => 'MyTable'
]);
```

7.4.7. Configure an auto-increment primary key column

This topic describes how to configure an auto-increment primary key column. You cannot set a partition key to an auto-increment column. If you write data to a table that contains an auto-increment primary key column, you do not need to specify specific values for the auto-increment primary key column because Tablestore generates values for the auto-increment primary key column. Values generated for the auto-increment primary key column are unique and consecutive within the partition that shares the same partition key value.

> **Note** Tablestore SDK for PHP V4.0.0 and later allow you to create auto-increment primary key columns.

Prerequisites

The OTSClient instance is initialized. For more information, see Initialization.

Procedure

1. When you create a table, you cannot set the partition key to an auto-increment primary key column.

   An auto-increment primary key column can only be an integer column. The generated value for an auto-increment primary key column is a 64-bit signed long integer.

2. When you write data to a table, you do not need to specify a specific value for the auto-increment column. Instead, you need only to set the value of the auto-increment column to placeholders.

   If you want to obtain the value of the auto-increment column after data is written to the table, you can set ReturnType to ReturnTypeConst::CONST_PK.

   When you query data, you must specify the values of all primary key columns. To obtain a complete primary key value, you can set ReturnType to ReturnTypeConst::CONST_PK in PutRow, UpdateRow, or BatchWriteRow.

Examples

When you use auto-increment primary key columns, you can call the CreateTable, PutRow, UpdateRow, or BatchWriteRow operation.

1. Create a table

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

```
function createTable($otsClient)
{
    $request = [
        'table_meta' => [
            'table_name' => 'table_name',       // Specify the table name.
            'primary_key_schema' => [
                ['PK_1', PrimaryKeyTypeConst::CONST_STRING],    // The name of the firs
t primary key column (partition key) is PK_1. The type of the first primary key column
is STRING.
                ['PK_2', PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyOptionConst::CON
ST_PK_AUTO_INCR]
                // The name of the second primary key column (partition key) is PK_2. T
he type of the second primary key column is INTEGER. Set this column to the primary key
column.
            ]
        ],
        'reserved_throughput' => [
            'capacity_unit' => [         // Set the reserved read throughput to 0 capac
ity units (CUs) and reserved write throughput to 0 CUs.
                'read' => 0,
                'write' => 0
            ]
        ],
        'table_options' => [
            'time_to_live' => -1,             // Specify that data never expires.
            'max_versions' => 1,              // Save only one version.
            'deviation_cell_version_in_sec' => 86400   // Set the max version offset of
data. Unit: seconds.
        ]
    ];
    $otsClient->createTable($request);
}
```

2. Write data

   When you write data to a table, you do not need to specify a value for the auto-increment column. Instead, you need only to set the value of the auto-increment column to PrimaryKeyTypeConst::CONST_PK_AUTO_INCR.

```
function putRow($otsClient)
{
    $row = [
        'table_name' => 'table_name',
        'primary_key' => [
            ['PK_1', 'Hangzhou'],                       // Set the primary key name and
value that are of the list type.
            ['PK_2', null, PrimaryKeyTypeConst::CONST_PK_AUTO_INCR]    // The auto-incr
ement primary key column. You do not need to specify the value. Instead, you need only
to set the value of the auto-increment column to PrimaryKeyTypeConst::CONST_PK_AUTO_INC
R. Tablestore automatically generates this value.
        ],
        'attribute_columns' => [              // Set the attribute column that is of th
e list type.
            ['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' => ReturnTypeConst::CONST_PK     // Set return_type to Return
TypeConst::CONST_PK to return the value of the auto-increment primary key column.
        ]
    ];
    $ret = $otsClient->putRow($row);
    print_r($ret);
    $primaryKey = $ret['primary_key'];                  // The obtained primary key val
ue can be passed to operations such as GetRow, UpdateRow, and DeleteRow.
    return $primaryKey;
}
```

7.4.8. Configure conditional update

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Parameters

You can use PutRow, UpdateRow, DeleteRow, and BatchWriteRow.for conditional update.

```
'condition' => [
    'row_existence' => <RowExistenceExpectation>
    'column_condition' => <ColumnCondition>
]
```

When only a row existence condition exists, it can be abbreviated:

```
    'condition' => <RowExistenceExpectation>
```

- The following code provides an example on how to construct SingleColumnValueCondition:

```
    [
        'column_name' => '<string>',
        'value' => <ColumnValue>,
        'comparator' => <ComparatorType>
        'pass_if_missing' => true || false
        'latest_version_only' => true || false
    ]
```

- The following code provides an example on how to construct CompositeColumnValueFilter:

```
    [
        'logical_operator' => <LogicalOperator>
        'sub_conditions' => [
            <ColumnCondition>,
            <ColumnCondition>,
            <ColumnCondition>,
            // other conditions
        ]
    ]
```

Parameter	Description
row_existence	When you modify a table, the system first checks the row existence condition. If the row existence condition is not met, the modification fails, and an error is reported. Row existence conditions include IGNORE, EXPECT_EXIST, and EXPECT_NOT_EXIST. In Tablestore, RowExistenceExpectationConst::CONST_IGNORE indicates IGNORE, RowExistenceExpectationConst::CONST_EXPECT_EXIST indicates EXPECT_EXIST, and RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST indicates EXPECT_NOT_EXIST. • IGNORE: No existence check is performed. • EXPECT_EXIST: The row is expected to exist. If the row exists, the condition is met. If the row does not exist, the condition is not met. • EXPECT_NOT_EXIST: Expect that the row does not exist. If the row does not exist, the condition is met. If the row exists, the condition is not met.
column_name	The column name.

Parameter	Description
value	The comparison value of the column. The parameter 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, the type must be specified. Otherwise, the type can be ignored. If the type is BINARY, you must set both the type and value.
comparator	The relational operator that is used to compare column values. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=). In Tablestore, ComparatorTypeConst::CONST_EQUAL indicates EQUAL (=), ComparatorTypeConst::CONST_NOT_EQUAL indicates NOT_EQUAL (!=), ComparatorTypeConst::CONST_GREATER_THAN indicates GREATER_THAN (>), ComparatorTypeConst::CONST_GREATER_EQUAL indicates GREATER_EQUAL (>=), ComparatorTypeConst::CONST_LESS_THAN indicates LESS_THAN (<), and ComparatorTypeConst::CONST_LESS_EQUAL indicates LESS_EQUAL (<=).
logical_operator	The logical operator that is used to combine multiple conditions. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. In Tablestore, LogicalOperatorConst::CONST_NOT indicates NOT, LogicalOperatorConst::CONST_AND indicates AND, and LogicalOperatorConst::CONST_OR indicates OR. Different logical operators require different number of subconditions. • When the logical operator is NOT, only one subcondition can be added. • When the logical operator is AND or OR, you must add at least two subconditions.
pass_if_missing	Specifies whether to pass the conditional check when a column does not exist in a row. The type of this parameter value is Boolean. The default value is true, which indicates that if the column does not exist in a row, the conditional check is passed, and the row meets the update conditions. When pass_if_missing is set to false, if the column does not exist in a row, the conditional check fails, and the row does not meet the update conditions.

Parameter	Description
latest_version_only	Specifies whether to use only the value of the latest version when the comparison value has multiple versions. The type of this parameter value is Boolean. The default value is true, which indicates that if the column has multiple versions of data, only the value of the latest version is used for comparison. When latest_version_only is set to false, the comparison values of all versions are used for comparison if the comparison value has multiple versions. In this case, if only the value of a version meets the condition, the conditional check is passed, and the row meets the update conditions.

Examples

- The following code provides an example on how to construct SingleColumnValueCondition:

```
    // Set conditional update. If the value of Col0 is 0, the conditional check is passed
.
    $column_condition = [
        'column_name' => 'Col0',
        'value' => 0,
        'comparator' => ComparatorTypeConst::CONST_EQUAL
        'pass_if_missing' => false                          // If a row does not contain t
he Col0 column, the conditional check fails.
        'latest_version_only' => true                       // Specify that only the lates
t version is used for comparison.
    ];
```

- The following code provides an example on how to construct CompositeColumnValueCondition:

```
// Set condition composite1 to (Col0 == 0) AND (Col1 > 100).
    $composite1 = [
        'logical_operator' => LogicalOperatorConst::CONST_AND,
        'sub_conditions' => [
            [
                'column_name' => 'Col0',
                'value' => 0,
                'comparator' => ComparatorTypeConst::CONST_EQUAL
            ],
            [
                'column_name' => 'Col1',
                'value' => 100,
                'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
            ]
        ]
    ];
    // Set condition composite2 to ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10).
    $composite2 = [
        'logical_operator' => LogicalOperatorConst::CONST_OR,
        'sub_conditions' => [
            $composite1,
            [
                'column_name' => 'Col2',
                'value' => 10,
                'comparator' => ComparatorTypeConst::CONST_LESS_EQUAL
            ]
        ]
    ];
```

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

```
    // Read a row of data.
    $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 increase the value of Col0 by 1.
    $request = [
        'table_name' => 'MyTable',
        'condition' => [
            'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST,
            'column_condition' => [                  // If the condition is met, the data
is updated.
                'column_name' => 'col0',
                'value' => $col0Value,
                'comparator' => ComparatorTypeConst::CONST_EQUAL
            ]
        ],
        'primary_key' => [ // The primary key.
            ['PK0', 123],
            ['PK1', 'abc']
        ],
        'update_of_attribute_columns'=> [
            'PUT' => [
                ['col0', $col0Value+1]
            ]
        ]
    ];
    $response = $otsClient->updateRow ($request);
```

7.4.9. Configure local transaction

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

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

You can use local transaction to specify that the operations on a partition key are atomic, which indicates that operations based on the specified partition key either succeed or fail. The isolation level of the local transaction is read committed.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.

- A data table is created. Data is written to the table.

Usage notes

1. Use startLocalTransaction to create a local transaction based on the specified partition key value and obtain the ID of the local transaction.
2. Read and write data within the local transaction.

   You can call the following operations on the local transaction: GetRow, PutRow, DeleteRow, UpdateRow, BatchWriteRow, and GetRange.
3. Use commitTransaction to commit local transactions or use abortTransaction to abort local transactions.

Limits

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

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

    If a transaction is not read or written for more than 60 seconds, the Tablestore server determines that the transaction times out and aborts the transaction.
  - Up to 4 MB of data can be written to each transaction. The volume of data written to each transaction is calculated in the same way as a regular write request.
  - If you do not specify a version number for a cell, the Tablestore server assigns a version number to the cell in the usual way when the cell is written to the transaction (rather than when the transaction is committed).
  - If a BatchWriteRow request includes a transaction ID, all rows in the request can be written only to the table that matches the transaction ID.
  - When you use a transaction, the data within the range specified based on the corresponding partition key value is locked. If a request that is sent to write data within the local transaction does not contain the transaction ID, the request fails. Data within the transaction is unlocked when the transaction is committed or aborted, or when the transaction times out.
  - A transaction remains valid even if a read or write request with the transaction ID is rejected. You can resend the request in the same manner as a regular request or you can abort the transaction.

API operations

- startLocalTransaction

```
    /**
     * Create a local transaction and obtain the local transaction ID.
     * @api
     * @param [] $request
     *           The request parameters, including the table name and partition key.
     * @return [] Response.
     * @throws OTSClientException The exception that is returned when a parameter error o
ccurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore serv
er returns an error.
     * @example "src/examples/StartLocalTransaction.php" 50
     */
    public function startLocalTransaction(array $request)
```

- commitTransaction

```
    /**
     * Commit a local transaction.
     * @api
     *
     * @param [] $request
     *          The request parameter, which is the transaction ID.
     * @return [] Response.
     * @throws OTSClientException The exception that is returned when a parameter error o
ccurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore serv
er returns an error.
     * @example "src/examples/CommitTransaction.php" 50
     */
    public function commitTransaction(array $request)
```

- abortTransaction

```
    /**
     * Abort a local transaction.
     * @api
     *
     * @param [] $request
     *          The transaction ID, which is the request parameter.
     * @return [] Response.
     * @throws OTSClientException The exception that occurs when a parameter error occurs
or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that occurs when the Tablestore server re
turns an error.
     * @example "src/examples/AbortTransaction.php" 20
     */
    public function abortTransaction(array $request)
```

Parameters

Parameter	Description
table_name	The name of the table.
key	The partition key of the table. When you create a local transaction, you need only to specify the partition key value for the local transaction.
primary_key	The primary key of the table. After a local transaction is created, you must specify a complete primary key when you read data from or write data to the local transaction.
transaction_id	The ID that identifies a local transaction. After a local transaction is created, the transaction ID must be included for operations on the local transaction.

Examples

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

```
$response = $this->otsClient->startLocalTransaction (array (
    'table_name' => 'TransactionTable',
    'key' => array(  // Primary keys: [PK0:INTEGER,PK1:STRING]
        array('PK0', 0)
    )
));
```

2. Read and write data within the local transaction.

   Local transaction IDs are used to read and write data within the local transaction, which is the same as the normal read and write operations on data.

```
$updateRequest = array(
    'table_name' => 'TransactionTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    '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);
```

3. Commit or abort the transaction.
   - Commit a local transaction for all modified data to take effect.

```
$this->otsClient->commitTransaction(array(
    'transaction_id' => $response['transaction_id']
));
```

- Abort a local transaction. Modifications to the data within the local transaction are ignored.

```
$this->otsClient->abortTransaction(array(
    'transaction_id' => $response['transaction_id']
));
```

7.4.10. Configure filter

Tablestore filters query results on the server side before returning the results. Only rows of data that match the filter conditions are returned.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

Usage notes

When you call the GetRow, BatchGetRow, or GetRange operation to query data, you can use a filter to return only the rows that meet the filter conditions.

The available filters are SingleColumnValueFilter, SingleColumnValueRegexFilter, and CompositeColumnValueFilter.

- SingleColumnValueFilter determines whether to filter a row based only on the values of a reference column.
- SingleColumnValueRegexFilter uses regular expressions to match column values of the String type and extract matching substrings. Then, this filter converts the data type of the extracted substrings to String, Integer, or Double and filters the values after conversion.
- CompositeColumnValueFilter determines whether to filter a row by combining the filter conditions of values of multiple reference columns.

> **Note** For more information about filters, see Configure a filter in Function Introduction.

Limits

- The filter conditions support relational operators such as =, !=, >, >=, <, and <=, and logical operators such as NOT, AND, and OR. You can specify a maximum of 10 conditions.
- Reference columns in filters must be included in the read result. If the specified columns from which data is read do not include reference columns, the filter cannot query the values of reference columns.
- You can use filters by calling the GetRow, BatchGetRow, and GetRange operations, which does not change the native semantics or limited items of these operations.

  When you use GetRange, the number of rows scanned cannot exceed 5,000, or the data scanned cannot exceed 4 MB in size.

If the scanned 5,000 rows or 4 MB does not match the filter conditions, the rows in the response are empty. However, next_start_primary_key may not be empty. In this case, you need to use next_start_primary_key to continue reading the data until next_start_primary_key is empty.

Parameters

Filters are applicable to the column_filter parameter of the GetRow, BatchGetRow, and GetRange operations.

```
    'column_filter' => <ColumnFilter>
```

- The following code provides an example on how to create SingleColumnValueFilter:

  SingleColumnValueFilter supports comparison between the value of a column that can be a primary key column and a constant. SingleColumnValueFilter does not support the comparison between two columns or two constants.

  ```
  [
      'column_name' => '<string>',
      'value' => <ColumnValue>,
      'comparator' => <ComparatorType>
      'pass_if_missing' => true || false
      'latest_version_only' => true || false
  ]
  ```

- The following code provides an example on how to create CompositeColumnValueFilter:

  CompositeColumnValueFilter has a tree structure. It contains the logical_operator inner node and the SingleColumnValueFilter leaf node.

  ```
  [
      'logical_operator' => <LogicalOperator>
      'sub_filters' => [
          <ColumnFilter>,
          <ColumnFilter>,
          <ColumnFilter>,
          // other conditions
      ]
  ]
  ```

Parameter	Description
column_name	The name of the reference column in the filter.

Parameter	Description
value	The comparison value of the reference column in the filter. The parameter 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, the type must be specified. Otherwise, the type can be ignored. If the type is BINARY, you must set both the type and value.
comparator	The relational operator in the filter. For more information, see ComparatorType. Relational operators include EQUAL (=), NOT_EQUAL (!=), GREATER_THAN (>), GREATER_EQUAL (>=), LESS_THAN (<), and LESS_EQUAL (<=). In Tablestore, ComparatorTypeConst::CONST_EQUAL indicates EQUAL (=), ComparatorTypeConst::CONST_NOT_EQUAL indicates NOT_EQUAL (!=), ComparatorTypeConst::CONST_GREATER_THAN indicates GREATER_THAN (>), ComparatorTypeConst::CONST_GREATER_EQUAL indicates GREATER_EQUAL (>=), ComparatorTypeConst::CONST_LESS_THAN indicates LESS_THAN (<), and ComparatorTypeConst::CONST_LESS_EQUAL indicates LESS_EQUAL (<=).
logical_operator	The logical operator in the filter. For more information, see LogicalOperator. Logical operators include NOT, AND, and OR. In Tablestore, LogicalOperatorConst::CONST_NOT indicates NOT, LogicalOperatorConst::CONST_AND indicates AND, and LogicalOperatorConst::CONST_OR indicates OR.
pass_if_missing	Specifies whether to return a row if the row does not contain the reference column. The type of this parameter value is Boolean. The default value is true, which indicates that if the reference column does not exist in a row, the row is returned. When pass_if_missing is set to false, if the reference column does not exist in a row, the row is not returned.
latest_version_only	Specifies whether to use only the value of the latest version for comparison when the reference column contains data of multiple versions. The type of this parameter value is Boolean. The default value is true, which indicates that if the reference column has multiple versions of data, only the value of the latest version is used for comparison. When latest_version_only is set to false, if a reference column has multiple versions of data, the value of all versions in the column is used for comparison. In this case, if only the value of one version meets the conditions, the row is returned.

Parameter	Description
sub_filters	Child nodes can be SingleColumnValueFilter or CompositeColumnValueFilter. Different logical operators of the inner nodes require different number of child nodes that can be attached. • When the logical operator of an inner node is NOT, only one child node can be attached. • When the logical operator of an inner node is AND or OR, multiple child nodes can be attached.

Examples

- The following code provides an example on how to construct SingleColumnValueFilter:

```
    // Specify that the row is returned when the value of Col0 is 0.
    $column_filter = [
        'column_name' => 'Col0',
        'value' => 0,
        'comparator' => ComparatorTypeConst::CONST_EQUAL
        'pass_if_missing' => false                          / If Col0 does not exist, the
row that contains this column is not returned.
        'latest_version_only' => true                       // Specify that only the lates
t version is used for comparison.
    ];
```

- The following code provides an example on how to construct CompositeColumnValueFilter:

```
// Set condition composite1 to (Col0 == 0) AND (Col1 > 100).
$composite1 = [
    'logical_operator' => LogicalOperatorConst::CONST_AND,
    'sub_filters' => [
        [
            'column_name' => 'Col0',
            'value' => 0,
            'comparator' => ComparatorTypeConst::CONST_EQUAL
        ],
        [
            'column_name' => 'Col1',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ]
    ]
];
// Set condition composite2 to ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10).
$composite2 = [
    'logical_operator' => LogicalOperatorConst::CONST_OR,
    'sub_filters' => [
        $composite1,
        [
            'column_name' => 'Col2',
            'value' => 10,
            'comparator' => ComparatorTypeConst::CONST_LESS_EQUAL
        ]
    ]
];
```

7.5. Single-row operations

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

PutRow

You can call this operation to insert a row. If the row exists, data of all versions in all columns of the row is deleted before new data is inserted.

- Operations

```
    /**
     * Write a row of data. If the row exists, data of all versions in all columns of the row is deleted before new data is inserted. The number of the consumed capacity units (CUs) is returned.
      * @api
     * @param [] $request The request parameters.
     * @return [] The response.
     * @throws OTSClientException The exception that is returned when a parameter error occurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore server returns an error.
     */
    public function putRow(array $request);
```

- Request parameters

Parameter	Description
table_name	The name of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update. ○ row_existence: the row existence condition. Note ■ RowExistenceExpectationConst::CONST_IGNORE indicates that new data is inserted into a row no matter whether the specified row exists. If the specified row exists, the existing data is overwritten. ■ RowExistenceExpectationConst::CONST_EXPECT_EXIST indicates that new data is inserted only when the specified row exists. The existing data is overwritten. ■ RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST indicates that data is inserted only when the specified row does not exist. ○ column_condition: the column-based condition.

Parameter	Description
primary_key	The primary key of the row. **Note** ◦ The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table. ◦ For an auto-increment primary key column, you need only to set the value of the auto-increment primary key column to a placeholder. For more information, see Configure an auto-increment primary key column. ◦ The primary key of a table can contain one to four primary key columns. Primary key columns are sorted in the order in which they are added. For example, the schema of PRIMARY KEY (A, B, C) is different from that of PRIMARY KEY (A, C, B). Tablestore sorts rows based on the values of all primary key columns. ◦ Each primary key column is specified by parameters in the following sequence: the primary key name, primary key value (PrimaryKeyValue), and primary key type (PrimaryKeyType). PrimaryKeyType is optional. ◦ The value of PrimaryKeyValue can be an integer, a binary, or a string. ◦ PrimaryKeyType can be set to PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyTypeConst::CONST_STRING, PrimaryKeyTypeConst::CONST_BINARY, or PrimaryKeyTypeConst::CONST_PK_AUTO_INCR (auto-increment primary key columns), which separately indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, and PK_AUTO_INCR (auto-increment primary key columns) types. If the type of data is INTEGER or STRING, this parameter can be ignored. Otherwise, the type must be specified.

Parameter	Description
attribute_columns	The attribute columns of the row. ○ An attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType, which is optional), and timestamp (optional). ○ ColumnType can be set to ColumnTypeConst::CONST_INTEGER, ColumnTypeConst::CONST_STRING, ColumnTypeConst::CONST_BINARY, ColumnTypeConst::CONST_BOOLEAN, or ColumnTypeConst::CONST_DOUBLE, which separately indicate the INTEGER, STRING (UTF-8 encoded string), BINARY, BOOLEAN, and DOUBLE types. If the type of data is BINARY, this parameter must be specified. Otherwise, the type can be ignored or set to null. ○ A timestamp is the data version number. For more information, see Data versions and TTL. You can customize a data version number or use the data version number generated by Tablestore. If you do not specify this parameter, the data version number generated by the system is used. ■ The version number generated by Tablestore is calculated based on the number of milliseconds that have elapsed since 00:00:00 UTC on January 1, 1970. ■ When you choose to specify the version number, make sure that the version number is a 64-bit timestamp accurate to the millisecond within the valid version range.
return_content	The return type. return_type: Set the value to ReturnTypeConst::CONST_PK, which returns the primary key values. This parameter is used for auto-increment primary key columns.

- Request format

```
$result = $client->putRow([
    'table_name' => '<string>', // Set the table name.
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                              // Set the primary key.
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => [         // Set the attribute column.
            ['<string>', <ColumnValue>],
            ['<string>', <ColumnValue>, <ColumnType>],
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
    ],
    'return_content' => [
        'return_type' => <ReturnType>
    ]
]);
```

- Response parameters

Parameter	Description
consumed	The number of CUs consumed by this operation. capacity_unit: the number of read and write CUs consumed. ○ read: the read throughput ○ write: the write throughput
primary_key	The value of the primary key, which is consistent with that in the request. **Note** If you set return_type to ReturnTypeConst::CONST_PK, the value of the primary key is returned.
attribute_columns	The values of attribute columns. The attribute column names are consistent with those specified in the request. At present, the value of this paramter is empty.

- Response format

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

- Example 1

  The following code provides an example on how to write a row that contains 10 attribute columns and write one version of data for each column. The version number (timestamp) is generated by Tablestore.

```
$attr = array();
for($i = 0; $i < 10; $i++) {
    $attr[] = ['Col'. $i, $i];
}
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE, // You can set the conditi
on to IGNORE, EXPECT_EXIST, or EXPECT_NOT_EXIST.
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

- Example 2

  The following code provides an example on how to write a row that contains 10 attribute columns and write three versions of data for each column. In this example, a custom version number (timestamp) is used.

```
$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, // You can set the conditi
on to IGNORE, EXPECT_EXIST, or EXPECT_NOT_EXIST.
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

- Example 3

  The following code provides an example on how to write a row that contains 10 attribute columns and write three versions of data for each column. In this example, a condition that expects the row not to exist is set and a custom version number (timestamp) is used.

```
$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 to write data when the expected row does not exist.
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

- Example 4

  The following code provides an example on how to write a row that contains 10 attribute columns and write three versions of data for each column. In this example, a condition that expects the row to exist and the Col0 value of the row to be greater than 100 is set and a custom version number (timestamp) is used.

```
$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 to write data when the expected 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' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

GetRow

You can call this operation to read a row of data.

The following results for the read request may be returned:

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

```
    /**
     * Read a row of data.
     * @api
     * @param [] $request The request parameters.
     * @return [] The response.
     * @throws OTSClientException The exception that is returned when a parameter error o
ccurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore serv
er returns an error.
     */
    public function getRow(array $request);
```

- Request parameters

Parameter	Description
table_name	The name of the table.
primary_key	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
max_versions	The maximum number of versions that can be read. **Note** You must specify at least one of max_versions and time_range. ◦ If you specify only max_versions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, data whose version number is within the specified range or a specified version of data is returned. ◦ If you specify both max_versions and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
time_range	Specifies a range of versions to read or a version of data to read. For more information, see TimeRange. **Note** You must specify at least one of max_versions and time_range. ◦ If you specify only max_versions, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, data whose version number is within the specified range or a specified version of data is returned. ◦ If you specify both max_versions and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ If you specify a range of versions to read, you must set start_time and end_time. start_time indicates the start timestamp. end_time indicates the end timestamp. The specified range includes the start timestamp and excludes the end timestamp. ◦ If you want to query data of a specified version, you must set specific_time. specific_time indicates a specified timestamp. You can set either of specific_time or [start_time, end_time). The timestamp used for the value of time_range ranges from 0 to INT64.MAX. Unit: milliseconds.
columns_to_get	The set of columns to read. The column name can be the primary key column or attribute column. If you do not specify this parameter, all data in the row is returned. **Note** ◦ By default, Tablestore returns the data from all columns of the row when you query a row. You can set the columns_to_get parameter to return the data in specified columns. For example, if you include col0 and col1 in columns_to_get, only the values of col0 and col1 are returned. ◦ If you use columns_to_get and column_filter together, Tablestore first queries the columns specified by columns_to_get, and then returns rows that meet the filter conditions.

Parameter	Description
start_column	The column from which you want to start reading. This parameter is used to read wide rows. The response contains the start column. The columns are sorted based on their names in alphabetical order. Example: 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 column at which you want to end reading. This parameter is used to read wide rows. The response does not contain the end column. The columns are sorted based on their names in alphabetical order. Example: 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	This parameter specifies the start position of a wide row in the next read operation. This feature is unavailable.
column_filter	You can set filter conditions to filter the queried results on the server side. Only rows that meet the specified filter conditions are returned. For more information, see Configure filter. **Note** If you use columns_to_get and column_filter together, Tablestore first queries the columns specified by columns_to_get, and then returns rows that meet the filter conditions.

- Request format

```
$result = $client->getRow([
    'table_name' => '<string>',                     // Set the table name.
    'primary_key' => [                              // Set the primary key.
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    '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>
]);
```

- Response parameters

Parameter	Description
consumed	The number of CUs consumed by this operation. capacity_unit: the number of read and write CUs consumed. ○ read: the read throughput ○ write: the write throughput
primary_key	The value of the primary key, which is consistent with that in the request. **Note** If the row does not exist, primary_key is an empty list [].

Parameter	Description
attribute_columns	The values of attribute columns. **Note** If the row does not exist, attribute_columns is an empty list []. ○ Each attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType), and timestamp. A timestamp is a 64-bit integer, which indicate multiple versions of data. ○ The attribute columns in the response are sorted by attribute column name in ascending alphabetical order. The versions of the attribute columns are sorted by timestamp in descending order. ○ The order of attribute columns may be different between the request and response.
next_token	The start position of a wide row in the next read operation. This feature is unavailable.

- Response format

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => [
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
    ],
    'next_token' => '<string>'
]
```

- Example 1

  The following code provides an example on how to read the latest versions of the specified columns from a row:

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'max_versions' => 1,                    // Set this parameter to read the latest ver
sion.
    'columns_to_get' => ['Col0']            // Set the column to read.
];
$response = $otsClient->getRow ($request);
```

- Example 2

  The following code provides an example on how to set filter conditions to read a row of data:

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'max_versions' => 1,                    // Set this parameter to read the latest ver
sion.
    'column_filter' => [                    // Set the filter to return a row if the val
ue of Col0 is 0.
        'column_name' => 'Col0',
        'value' => 0,
        'comparator' => ComparatorTypeConst::CONST_EQUAL,
        'pass_if_missing' => false          // If the Col0 column does not exist, the row
is not returned.
    ]
];
$response = $otsClient->getRow ($request);
```

UpdateRow

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

> **Note** If the UpdateRow request contains only columns to delete, and the specified row does not exist, a new row is not added.

- Operations

```
    /**
     * Update a row of data.
     * @api
     * @param [] $request The request parameters.
     * @return [] The response.
     * @throws OTSClientException The exception that is returned when a parameter error o
ccurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore serv
er returns an error.
     */
    public function updateRow(array $request);
```

- Request parameters

Parameter	Description
table_name	The name of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.
primary_key	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
update_of_attribute_columns	The attribute column to be updated.
return_content	The return type. return_type: Set the value to ReturnTypeConst::CONST_PK, which returns the primary key values. This parameter is used for auto-increment primary key columns.

- Request format

```
$result = $client->updateRow([
    'table_name' => '<string>', // Set the table name.
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                              // Set the primary key.
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'update_of_attribute_columns' => [         // Set the attribute column to update.
        '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>
    ]
]);
```

- Response parameters

Parameter	Description
consumed	The number of CUs consumed by this operation. capacity_unit: the number of read and write CUs consumed. ○ read: the read throughput ○ write: the write throughput

Parameter	Description
primary_key	The value of the primary key, which is consistent with that in the request. **Note** If you set return_type to ReturnTypeConst::CONST_PK, the value of the primary key is returned.
attribute_columns	The values of attribute columns. The attribute column names are consistent with those specified in the request. At present, the value of this paramter is empty.

- Response format

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

- Example 1

  The following code provides an example on how to update several columns, delete the specified version of a column, and delete the specified column:

```
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => [ // Set 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, 1526418378526]
        ],
        'DELETE' => [                   // Delete a version of a column.
            ['Col10', 1526418378526]
        ],
        'DELETE_ALL' => [
            'Col11'                     // Delete a column.
        ]
    ]
];
$response = $otsClient->updateRow($request);
```

- Example 2

  The following code provides an example on how to configure update conditions:

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST, // Update th
e row if it exists.
        'column_filter' => [                                                 // Update th
e 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, 1526418378526]
        ],
        'DELETE' => [                   // Delete a version of a column.
            ['Col10', 1526418378526]
        ],
        'DELETE_ALL' => [
            'Col11'                     // Delete a column.
        ]
    ]
];
```

DeleteRow

You can call this operation to delete a row. If the row you want to delete does not exist, no changes are made to the table.

- Operations

```
    /**
     * Delete a row of data.
     * @api
     * @param [] $request The request parameters.
     * @return [] The response.
     * @throws OTSClientException The exception that is returned when a parameter error o
ccurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore serv
er returns an error.
     */
    public function deleteRow(array $request);
```

- Request parameters

Parameter	Description
table_name	The name of the table.
condition	You can use conditional update to set a row existence condition or columns-based conditions for the row. For more information, see Configure conditional update.
primary_key	The primary key of the row. **Note** The configured number and types of primary key columns must be consistent with the actual number and types of primary key columns of the table.
return_content	The return type. return_type: Set the value to ReturnTypeConst::CONST_PK, which returns the primary key values. This parameter is used for auto-increment primary key columns.

- Request format

```
$result = $client->deleteRow([
    'table_name' => '<string>', // Set the table name.
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                              // Set the primary key.
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'return_content' => [
        'return_type' => <ReturnType>
    ]
]);
```

- Response parameters

Parameter	Description
consumed	The number of CUs consumed by this operation. capacity_unit: the number of read and write CUs consumed. ○ read: the read throughput ○ write: the write throughput

Parameter	Description
primary_key	The value of the primary key, which is consistent with that in the request. **Note** If you set return_type to ReturnTypeConst::CONST_PK, the value of the primary key is returned.
attribute_columns	The values of attribute columns. The attribute column names are consistent with those specified in the request. At present, the value of this paramter is empty.

- Response format

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

- Example 1

  The following code provides an example on how to delete a row from a table:

```
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'return_content' => [
        'return_type' => ReturnTypeConst::CONST_PK     // Set return_type to return the v
alue of the primary key for auto-increment primary key columns.
    ]
];
$response = $otsClient->deleteRow($request);
```

- Example 2

  The following code provides an example on how to set deletion conditions:

```
$request = [
    'table_name' => 'MyTable',
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST, // Delete th
e row if it exists.
        'column_filter' => [          // Delete the row when the value of Col0 is greater
than 100.
            'column_name' => 'Col0',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ],
    ],
    'primary_key' => [ // Set the primary key.
        ['PK0', 123],
        ['PK1', 'abc']
    ]
];
$response = $otsClient->deleteRow ($request);
```

7.6. Multi-row operations

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

BatchWriteRow

You can call this operation to write several rows of data to one or more tables by sending one request. The BatchWriteRow operation consists of PutRow, UpdateRow, and DeleteRow operations. The process of constructing a suboperation is the same as that of calling the PutRow, UpdateRow, and DeleteRow operations. BatchWriteRow supports conditional updates.

Each suboperation of BatchWriteRow is independently performed, and the result for each suboperation is independently returned.

- When multiple rows are written, part of rows may fail to be written. If the operation fails for part of rows, Tablestore does not return exceptions. However, information about the index and error messages of the failed rows is contained in BatchWriteRowResponse. Therefore, when you call the BatchWriteRow operation, you must check the returned row to determine whether the status of each row is successful. If you do not check the returned value, failures of operations on part of rows are ignored.
- When the server detects incorrect parameters during operations, BatchWriteRow may return a parameter exception when none of the operations are performed for the request.
- Operations

```
/**
 * Write, update, or delete the specified rows of data.
 * Be aware that if the BatchWriteRow operation fails for part of rows, the system does n
ot return an exception. However, information about the failed rows is contained in $respo
nse. For more information, see the sample response to the BatchWriteRow operation.
 * @api
 * @param [] $request The request parameters.
 * @return [] The response.
 * @throws OTSClientException The exception that occurs when a parameter error occurs or
the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that occurs when the Tablestore server return
s an error.
 */
public function batchWriteRow(array $request);
```

- Request parameters

  BatchWriteRow is a combination of PutRow, UpdateRow, and DeleteRow.

  - Hierarchies are created for tables. Multiple tables can be processed at a time.

    The values of the tables parameter are based on tables. This parameter specifies information about tables and rows on which the write, modification, or delete operations are performed. For more information about parameters, see Single-row operations.

  - The operation_type parameter is added to distinguish between different operation types.

    You can use OperationTypeConst::CONST_PUT to set the operation type to PUT, OperationTypeConst::CONST_UPDATE to UPDATE, and OperationTypeConst::CONST_DELETE to DELETE.

    - 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.

- Request format

```
$result = $client->batchWriteRow([
    'tables' => [                                           // Set the hierarchies of ta
bles.
        [
            'table_name' => '<string>',                     // Set the name of the table
.
            'operation_type' => <OperationType>,
            'condition' => [
                'row_existence' => <RowExistence>,
                'column_condition' => <ColumnCondition>
            ],
            'primary_key' => [                              // Set the primary key.
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'attribute_columns' => [                        // This parameter is required
when the operation type is PUT.
                    ['<string>', <ColumnValue>],
                    ['<string>', <ColumnValue>, <ColumnType>],
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ],
            'update_of_attribute_columns' => [              // This parameter is require
d when the operation type is UPDATE.
                '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>
            ]
        ],
        // Set other tables.
    ]
]);
```

- Response parameters

  Each table in the tables parameter in the response corresponds to the table specified in the request. The following table describes the parameters.

Parameter	Description
table_name	The name of the table.
is_ok	Indicates whether the operation for this row was successful. ○ A value of true indicates that the row is written. In this case, the error parameter is invalid. ○ A value of false indicates that the row fails to be written.
error	The error information in the response when the operation failed. ○ The error code in the response when the operation on the single row failed. ○ The error message in the response when the operation on the single row failed.
consumed	The number of capacity units (CUs) consumed by this operation. capacity_unit: the number of read and write CUs consumed. ○ read: the read throughput ○ write: the write throughput
primary_key	The value of the primary key, which is consistent with that in the request. If return_type is set to RT_PK, values of all primary key columns are returned. return_type is used for auto-increment of primary key columns.
attribute_columns	The values of attribute columns. The attribute column names are consistent with those specified in the request. At present, the value of this parameter is empty.

- Response format

```
[
    'tables' => [
        [
            'table_name' => '<string>',
            'rows' => [
                [
                    'is_ok' => true || false,
                    'error' => [
                        'code' => '<string>',
                        'message' => '<string>',
                    ]
                    'consumed' => [
                        'capacity_unit' => [
                            'read' => <integer>,
                            'write' => <integer>
                        ]
                    ],
                    'primary_key' => [
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
                    ],
                    'attribute_columns' => []
                ],
                // Set other rows.
            ]
        ],
        // Set other tables.
    ]
]
```

- Examples

  The following code provides an example on how to write 30 rows of data to three tables at a time. Each table is written with 10 rows.

```
// Write data to three tables. Each table is written with 10 rows.
$tables = array();
for($i = 0; $i < 3; $i++) {
    $rows = array();
    for($j = 0; $j < 10; $j++) {
        $rows[] = [
            'operation_type' => OperationTypeConst::CONST_PUT, // Set the operation type
to PUT.
            'condition' => RowExistenceExpectationConst::CONST_IGNORE,
            'primary_key' => [
                ['pk0', $i],
                ['pk1', $j]
            ],
            'attribute_columns' => [
                ['Col0', 4],
                ['Col2', 'Beijing']
            ]
        ];
    }
    $tables[] = [
        'table_name' => 'SampleTable' . $i,
        'rows' => $rows
    ];
}
$request = [
    'tables' => $tables
];
$response = $otsClient->batchWriteRow ($request);
// Process each returned table.
foreach ($response['tables'] as $tableData) {
  print "Handling table {$tableData['table_name']} ...\n";
  // Process the result returned by PutRow on the table.
  $putRows = $tableData['rows'];
  foreach ($putRows as $rowData) {
    if ($rowData['is_ok']) {
      // Data is written.
      print "Capacity Unit Consumed: {$rowData['consumed']['capacity_unit']['write']}\n";
    } else {
      // Handle errors.
      print "Error: {$rowData['error']['code']} {$rowData['error']['message']}\n";
    }
  }
}
```

The following table lists the files that contain detailed sample code.

File	Description
BatchWriteRow1.php	Shows you how to perform multiple PUT operations in BatchWriteRow.
BatchWriteRow2.php	Shows you how to perform multiple UPDATE operations in BatchWriteRow.

File	Description
BatchWriteRow3.php	Shows you how to perform multiple DELETE operations in BatchWriteRow.
BatchWriteRow4.php	Shows you how to perform the UPDATE, PUT, and DELETE operations in BatchWriteRow.
BatchWriteRowWithColumnFilter.php	Shows you how to use BatchWriteRow with conditional update.

BatchGetRow

You can call this operation to read several rows of data from one or more tables by sending one request. The BatchGetRow operation consists of multiple GetRow operations. The process of constructing a suboperation is the same as that of calling the GetRow operation. BatchGetRow supports filters.

Be aware that BatchGetRow uses the same parameter configurations for all rows. For example, if ColumnsToGet is set to colA, only the value of colA is read for all the rows.

The execution and returning of results of each suboperation are carried out independently.

When multiple rows are read, part of rows may fail to be read. If the operation fails for part of rows, the system does not return exceptions. However, information about the error messages of the failed rows is stored in BatchGetRowResponse. Therefore, when you call the BatchGetRow operation, you must check the returned values to determine whether the status of each row is successful.

- Operations

```
/**
 * Read the specified rows of data.
 * Be aware that if the BatchGetRow operation fails for part of rows, Tablestore does not
return an exception. However, information about the failed rows is contained in $response
. For more information, see the sample response to the BatchGetRow operation.
 * @api
 * @param [] $request The request parameters.
 * @return [] The response.
 * @throws OTSClientException The exception that occurs when a parameter error occurs or
the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that occurs when the Tablestore server return
s an error.
 */
public function batchGetRow(array $request);
```

- Request parameters

  BatchGetRow and GetRow have the following differences:

  - Hierarchies are created for tables. Data from multiple tables can be read at a time.

    The values of the tables parameter are based on tables. This parameter specifies information about tables and rows on which various operations are performed. For more information about parameters, see Single-row operations.

- primary_key is changed to primary_keys, which allows you to set primary keys for multiple rows and read data from multiple rows at a time.

- Request format

```
$result = $client->batchGetRow([
    'tables' => [                                           // Set the hierarchies of ta
bles.
        [
            'table_name' => '<string>',                     // Set the name of the table
.
            'primary_keys' => [                             // Set the primary key.
                [
                    ['<string>', <PrimaryKeyValue>],
                    ['<string>', <PrimaryKeyValue>],
                    ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
                ],
                // Set 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>
        ],
        // Set other data tables.
    ]
]);
```

- Response parameters

  Each table in the tables parameter in the response corresponds to the table specified in the request. The following table describes the parameters.

Parameter	Description
table_name	The name of the table.
is_ok	Indicates whether the operation for this row was successful. ○ A value of true indicates that the row is read. In this case, the error parameter is invalid. ○ A value of false indicates that the row fails to be read. In this case, consumed, primary_key, and attribute_columns become invalid.

Parameter	Description
error	The error information in the response when the operation failed. ○ The error code in the response when the operation on the single row failed. ○ The error message in the response when the operation on the single row failed.
consumed	The number of CUs consumed by this operation. capacity_unit: the number of read and write CUs consumed. ○ read: the read throughput ○ write: the write throughput
primary_key	The value of the primary key, which is consistent with that in the request. **Note** If the row does not exist, primary_key is an empty list [].
attribute_columns	The values of attribute columns. **Note** If the row does not exist, attribute_columns is an empty list []. ○ Each attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType), and timestamp. A timestamp is a 64-bit integer, which indicates a version of data. ○ The attribute columns in the response are sorted by attribute column name in ascending alphabetical order. The versions of the attribute columns are sorted by timestamp in descending order. ○ The order of attribute columns may be different between the request and response.
next_token	The start position of a wide row in the next read operation. This feature is unavailable.

- Response format

```
[
    'tables' => [
        [
            'table_name' => '<string>',
            'rows' => [
                [
                    'is_ok' => true || false,
                    'error' => [
                        'code' => '<string>',
                        'message' => '<string>',
                    ]
                    'consumed' => [
                        'capacity_unit' => [
                            'read' => <integer>,
                            'write' => <integer>
                        ]
                    ],
                    'primary_key' => [
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
                    ],
                    'attribute_columns' => [
                            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                    ],
                    'next_token' => '<string>'
                ],
                // Set other rows.
            ]
        ],
        // Set other tables.
    ]
]
```

- Examples

  The following code provides an example on how to read 30 rows of data from three tables at a time. 10 rows of data are read from each table.

```
// Read data from three tables. Read 10 rows of data from each table.
$tables = array();
for($i = 0; $i < 3; $i++) {
    $primary_keys = array();
    for($j = 0; $j < 10; $j++) {
        $primary_keys[] = [
            ['pk0', $i],
            ['pk1', $j]
        ];
    }
    $tables[] = [
        'table_name' => 'SampleTable' . $i,
        'max_versions' => 1,
        'primary_keys' => $primary_keys
    ];
}
$request = [
    'tables' => $tables
];
$response = $otsClient->batchGetRow ($request);
// Process each returned table.
foreach ($response['tables'] as $tableData) {
  print "Handling table {$tableData['table_name']} ...\n";
  // Process each row of data in the table.
  foreach ($tableData['rows'] as $rowData) {
    if ($rowData['is_ok']) {
      // Process the data that is read.
        $row = json_encode($rowData['primary_key']);
        print "Handling row: {$row}\n";
    } else {
      // Handle errors.
      print "Error: {$rowData['error']['code']} {$rowData['error']['message']}\n";
    }
  }
}
```

The following table lists the files that contain detailed sample code.

File	Description
BatchGetRow1.php	Shows you how to use BatchGetRow to read multiple rows from a table.
BatchGetRow2.php	Shows you how to use BatchGetRow to read multiple rows from multiple tables.
BatchGetRow3.php	Shows you how to use BatchGetRow to read specified columns for multiple rows from a table.
BatchGetRow4.php	Shows you how to use BatchGetRow to process returned results.
BatchGetRowWithColumnFilter.php	Show you how to use BatchGetRow with filters.

GetRange

You can call this operation to read rows of data in a forward or backward direction based on a specified primary key range.

You can also set the number of rows you want 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 the next primary key information are returned. You can also initiate a request to start from where the last operation left off and read the remaining rows based on the next primary key information returned by the previous operation.

The GetRange operation may stop and return data in the following situations:

- The amount of read data reaches 4 MB.
- The number of read rows reaches 5,000.
- The number of returned rows reaches the limit.
- All reserved read throughput is consumed. You have no reserved read throughput to read the next row of data.

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

- Operations

```
/**
 * Read data between the start and the end primary key columns.
 * Be aware that the server may truncate this range. You need to determine whether to cal
l the GetRange operation again based on the next_start_primary_key value in the response.
 * You can specify the maximum number of rows to read.
 * When you specify the start and end primary key columns, you can use INF_MIN to indicat
e the minimum value and INF_MAX the maximum value. For more information, see the followin
g code samples:
 * @api
 * @param [] $request The request parameters.
 * @return [] The response.
 * @throws OTSClientException The exception that occurs when a parameter error occurs or
the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that occurs when the Tablestore server return
s an error.
 */
public function getRange(array $request);
```

- Parameters

  GetRange is different from GetRow in the following items:

  - The primary_key parameter is changed to the inclusive_start_primary_key and exclusive_end_primary_key parameters. The range is a left-closed and right-open interval.
  - The direction parameter is added to specify the direction for reading.
  - The limit parameter is added to limit the number of returned rows.

Parameter	Description
table_name	The name of the table.
inclusive_start_primary_key exclusive_end_primary_key	The start and end primary key columns of the read operation. The start and end primary key columns must be valid primary key columns or virtual points of the INF_MIN or INF_MAX type. The number of columns for the virtual point must be the same as that of the primary key. INF_MIN indicates an infinitely small value of which all other values of other types are larger. INF_MAX indicates an infinitely large value of which all other values of other types are smaller. ◦ inclusive_start_primary_key specifies the start primary key. If the row exists, the response contains this row. ◦ exclusive_end_primary_key specifies the end primary key. The response excludes the row no matter whether the row exists. Rows in tables are sorted based on the primary key column values in ascending order. The range for reading is a left-closed and right-open interval. When data is read in the forward direction, all rows whose primary key column values are greater than or equal to the start primary key column value and smaller than the end primary key column value are returned.
direction	The direction in which to read data. ◦ If DirectionConst::CONST_FORWARD is set, the start primary key column value must be smaller than the end primary key column value. The rows are returned based on the primary key column values in ascending order. ◦ If DirectionConst::CONST_BACKWARD is set, the start primary key column value must be greater than the end primary key column value. The rows are returned based on the primary key column values in descending order. Example: A table contains two primary keys A and B where the value of A is smaller than that of B. If you set direction to forward, the rows whose primary key values are greater than or equal to the value of A and smaller than the value of B are returned in ascending order. If you set direction to backward, the rows whose primary key values are greater than or equal to the value of B and smaller than the value of A are returned in descending order.
limit	The maximum number of rows that can be returned. The value of this parameter must be greater than 0. An operation stops when the maximum number of rows are returned in the forward or backward direction, even if part of rows within the specified range are not returned. At this time, you can record the next_start_primary_key value in the response for the next reading.

Parameter	Description
max_versions	The maximum number of versions that can be read. **Note** You must specify at least one of max_version and time_range. ◦ If you specify only max_version, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, data whose version number is within the specified range or a specified version of data is returned. ◦ If you specify both max_version and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest.

Parameter	Description
time_range	Specifies a range of versions to read or a version of data to read. For more information, see TimeRange. **Note** You must specify at least one of max_version and time_range. ◦ If you specify only max_version, data of up to the specified number of versions is returned from the latest to the earliest. ◦ If you specify only time_range, data whose version number is within the specified range or a specified version of data is returned. ◦ If you specify both max_version and time_range, data of up to the specified number of versions within the time range is returned from the latest to the earliest. ◦ If you specify a range of versions to read, you must set start_time and end_time. start_time indicates the start timestamp. end_time indicates the end timestamp. The specified range includes the start timestamp and excludes the end timestamp. ◦ If you want to query data of a specified version, you must set specific_time. specific_time indicates a specified timestamp. You can set either of specific_time or [start_time, end_time). The timestamp in the value of time_range ranges from 0 to INT64.MAX.

Parameter	Description
columns_to_get	The names of the columns to read. You can specify the names of primary key columns and the names of attribute columns. If you do not specify this parameter, all data in the row is returned. **Note** ◦ By default, Tablestore returns the data from all columns of the row when you query a row. You can set the columns_to_get parameter to return the data in specified columns. For example, if you include col0 and col1 in columns_to_get, only the values of col0 and col1 are returned. ◦ If a row is within the specified range of primary key values to read but does not contain the specified columns to return, the row is not included in the response. ◦ If you use columns_to_get and column_filter together, Tablestore first queries the columns specified by columns_to_get, and then returns rows that meet the filter conditions.
start_column	The column from which you want to start reading. This parameter is used to read wide rows. The response contains the current start column. The columns are sorted based on their names in alphabetical order. Example: 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 column at which you want to end reading. This parameter is used to read wide rows. The response does not contain the current end column. The columns are sorted based on their names in alphabetical order. Example: 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 start position of a wide row in the next read operation. This feature is unavailable.

Parameter	Description
column_filter	You can set filter conditions to filter the queried results on the server side. Only rows that meet the specified filter conditions are returned. For more information, see Filter. **Note** If you use columns_to_get and column_filter together, Tablestore first queries the columns specified by columns_to_get, and then returns rows that meet the filter conditions.

- Request format

```
$result = $client->getRange([
    'table_name' => '<string>',                                     // Set the name of th
e table.
    'inclusive_start_primary_key' => [                              // Set the start prim
ary key.
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'exclusive_end_primary_key' => [                                // Set the end primar
y key.
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'direction' => <Direction>,                                     // Set the direction
for reading.
    '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>
]);
```

- Response parameters

Parameter	Description
consumed	The number of CUs consumed by this operation. capacity_unit: the number of read and write CUs consumed. ○ read: the read throughput ○ write: the write throughput
primary_key	The value of the primary key, which is consistent with that in the request.
attribute_columns	The values of attribute columns. ○ Each attribute column is specified by parameters in the following sequence: the attribute column name, attribute column value (ColumnValue), attribute column value type (ColumnType), and timestamp. A timestamp is a 64-bit integer, which indicates a version of data. Unit: milliseconds. ○ The attribute columns in the response are sorted by attribute column name in ascending alphabetical order. The versions of the attribute columns are sorted by timestamp in descending order. ○ The order of attribute columns may be different between the request and response.
next_start_primary_key	The start primary key for the next reading. The value of next_start_primary_key can be used to determine whether all data is read. ○ If the value of next_start_primary_key is not empty in the response, the next_start_primary_key value can be used as the start primary key for the next GetRange operation. ○ If the value of next_start_primary_key is empty in the response, all data within the range is returned.

- Response format

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'next_start_primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'rows' => [
        [
            'primary_key' => [
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'attribute_columns' => [
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ]
        ],
        // Set other rows.
    ]
]
```

- Examples

  The following code provides an example on how to read data within a specified range:

```
// Search for rows whose PK0 value is within [1, 4).
// You must provide complete primary key columns to set the range. If the query range doe
s not involve the range of a column value, set the column value to an infinitely great (I
NF_MAX) or small (INF_MIN) value.
$startPK = [
    ['PK0', 1],
    ['PK1', null, PrimaryKeyTypeConst::CONST_INF_MIN]  // INF_MIN indicates the minimum v
alue.
];
// You must provide complete primary key columns to set the range. If the query range doe
s not involve the range of a column value, set the column value to an infinitely great (I
NF_MAX) or small (INF_MIN) value.
$endPK = [
    ['PK0', 4],
    ['PK1', null, PrimaryKeyTypeConst::CONST_INF_MAX]  // INF_MAX indicates the maximum v
alue.
];
$request = [
    'table_name' => 'SampleTable',
    'max_versions' => 1,                          // Set this parameter to read the lates
t version.
    'direction' => DirectionConst::CONST_FORWARD, // Query data in the forward direction.
    'inclusive_start_primary_key' => $startPK,    // Set the start primary key.
    'exclusive_end_primary_key' => $endPK,        // Set the end primary key.
    'limit' => 10                                 // Specify that a maximum of 10 columns
are returned.
];
$response = $otsClient->getRange ($request);
print "Read CU Consumed: {$response['consumed']['capacity_unit']['read']}\n";
foreach ($response['rows'] as $rowData) {
  // Process each row of data.
}
```

The following table lists the files that contain detailed sample code.

File	Description
GetRange1.php	Shows you how to use GetRange.
GetRange2.php	Shows you how to use GetRange to obtain the specified columns.
GetRange3.php	Shows you how to use GetRange to obtain the specified number of rows.
GetRangeWithColumnFilter.php	Shows you how you use GetRange with filter.

7.7. 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' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
    array(
        'pk_sort' => array(
            'order' => SortOrderConst::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' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
    array(
        'pk_sort' => array(
            'order' => SortOrderConst::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_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 10,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        '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_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::FUNCTION_SCORE_QUERY,
            'query' => array(
                'query' => array(
                    'query_type' => QueryTypeConst::TERM_QUERY,
                    'query' => array(
                        'field_name' => 'keyword',
                        'term' => 'keyword'
                    )
                ),
                'field_value_factor' => array(
                    'field_name' => 'long'
                )
            )
        ),
        'sort' => array(
            array(
                'score_sort' => array(
                    'order' => SortOrderConst::SORT_ORDER_DESC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::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 sp
ecify the sorting method because the token includes the sorting information.
    $response = $otsClient->search($request);
    print json_encode($response['rows'], JSON_PRETTY_PRINT);
}
```

7.7.1. Create search indexes

Prerequisites

- OTSClient is initialized. For more information, see Initialization.
- A data table whose time_to_live is set to -1 and max_versions is set to 1 is created.

Operation

```
/**
 * Create a search index.
 * @api
 *
 * @param [] $request
 *            The request parameters, including the table name and index configuration.
 * @return [] The response.
 * @throws OTSClientException The exception that is returned when a parameter error occurs
or the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that is returned when the Tablestore server ret
urns an error.
 * @example "src/examples/CreateSearchIndex.php"
 */
public function createSearchIndex(array $request)
```

Parameters

When you create a search index, you must specify table_name, index_name, and schema. You must specify field_schemas, index_setting, and index_sort in schema. The following table describes the parameters.

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.

Parameter	Description
field_schemas	The list of field schemas. Each field schema contains the following parameters: • field_name: required. This parameter specifies the name of the field in the search index. The value is a column name. Type: String. A field in a search index can be a primary key column or an attribute column. • field_type: required. This parameter specifies the type of the field. Use FieldTypeConst::XXX to set the type. For more information, see Data types of column values. • is_array: optional. This parameter specifies whether the value is an array. Type: Boolean. If you set this parameter to true, the column stores data as an array. Data written to the column must be a JSON array. Example: ["a","b","c"]. Nested values are an array. If you set field_type to Nested, skip this parameter. • index: optional. This parameter specifies whether to create an index for the column. Type: Boolean. By default, this parameter is set to true, and Tablestore creates an inverted index or spatial index for the column. If this parameter is set to false, Tablestore does not create indexes for the column. • analyzer: optional. This parameter specifies the type of analyzer to use. If field_type is set to Text, you can set this parameter. If you do not specify this parameter, single-word tokenization is used. For more information about tokenization, see Tokenization. • enable_sort_and_agg: optional. This parameter specifies whether to enable the sorting and aggregation features. Type: Boolean. Before you enable the sorting feature, you need to set enable_sort_and_agg to true for a field. For more information about sorting, see Sorting and pagination. Notice • store: optional. This parameter specifies whether to store the value of the field in the search index. Type: Boolean.
index_setting	The settings of the search index, including routing_fields. routing_fields: optional. This parameter specifies custom routing fields. You can specify some primary key columns as routing fields. Tablestore 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.

Parameter	Description
index_sort	The presorting settings of the search index, including sorters. If index_sort is left empty, data is sorted by primary key. **Note** You can skip the presorting settings for search indexes that contain the Nested field type. sorters: required. This parameter specifies the presorting method for the search index. PrimaryKeySort and FieldSort are supported. For more information about sorting, see Sorting and pagination. • PrimaryKeySort: Data is sorted by primary key. You can configure the following parameter for PrimaryKeySort: order: the sort order. Data can be sorted in ascending or descending order. Default value: SortOrderConst::SORT_ORDER_ASC. • FieldSort: Data is sorted by field value. You can configure the following parameters for FieldSort: You can presort field values only when a search index is created and the sorting and aggregation features are enabled for fields in the search index. ◦ field_name: the name of the field to sort. ◦ order: the sort order. Data can be sorted in ascending or descending order. Default value: SortOrderConst::SORT_ORDER_ASC. ◦ mode: the sorting method used when the field has multiple values.

Examples

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'schema' => array(
        'field_schemas' => array(
            array(
                'field_name' => 'keyword',
                'field_type' => FieldTypeConst::KEYWORD,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'text',
                'field_type' => FieldTypeConst::TEXT,
                'analyzer' => 'single_word',
                'index' => true,
                'enable_sort_and_agg' => false,
                'store' => true,
                'is_array' => false
            ),
            array(
```

```
                'field_name' => 'geo',
                'field_type' => FieldTypeConst::GEO_POINT,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'long',
                'field_type' => FieldTypeConst::LONG,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'double',
                'field_type' => FieldTypeConst::DOUBLE,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'boolean',
                'field_type' => FieldTypeConst::BOOLEAN,
                'index' => true,
                'enable_sort_and_agg' => false,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'array',
                'field_type' => FieldTypeConst::KEYWORD,
                'index' => true,
                'enable_sort_and_agg' => false,
                'store' => true,
                'is_array' => true
            ),
            array(
                'field_name' => 'nested',
                'field_type' => FieldTypeConst::NESTED,
                'index' => false,
                'enable_sort_and_agg' => false,
                'store' => false,
                'field_schemas' => array(
                    array(
                        'field_name' => 'nested_keyword',
                        'field_type' => FieldTypeConst::KEYWORD,
                        'index' => false,
                        'enable_sort_and_agg' => false,
                        'store' => false,
                        'is_array' => false
                    )
```

```
                )
            ),
        ),
        'index_setting' => array(
            'routing_fields' => array("pk1")
        ),
//        "index_sort" => array(// You can skip the presorting settings for search indexes
that contain the Nested field type.
//            array(
//                'field_sort' => array(
//                    'field_name' => 'keyword',
//                    'order' => SortOrderConst::SORT_ORDER_ASC,
//                    'mode' => SortModeConst::SORT_MODE_AVG,
//                )
//            ),
//            array(
//                'pk_sort' => array(
//                    'order' => SortOrderConst::SORT_ORDER_ASC
//                )
//            ),
//        )
    )
);
$response = $otsClient->createSearchIndex();
```

7.7.2. Delete search indexes

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Operations

```
/**
 *Delete the search index.
 * @api
 *
 * @param [] $request
 *            The request parameters, including the table name and search index name.
 * @return [] The response.
 * @throws OTSClientException The exception that is returned when a parameter error occurs
or the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that is returned when the Tablestore server ret
urns an error.
 * @example "src/examples/DeleteSearchIndex.php"
 */
public function deleteSearchIndex(array $request)
```

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.

Examples

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index'
);
$response = $otsClient->deleteSearchIndex($request);
```

7.7.3. 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.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Operations

```
    /**
     * Obtain the description of a search index associated with the table.
     * @api
     *
     * @param [] $request
     *            The request parameter, which is the table name.
     * @return [] The response.
     * @throws OTSClientException The exception that is returned when a parameter error occ
urs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore server
returns an error.
     * @example "src/examples/DescribeSearchIndex.php" 20
     */
    public function describeSearchIndex(array $request)
```

Parameters

Parameter	Description
table_name	The name of the table.

Parameter	Description
index_name	The name of the search index.

Examples

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index'
);
$response = $otsClient->describeSearchIndex($request);
```

7.7.4. List search indexes

You can call the ListSearchIndex operation to obtain indexes created in the current instance or associated with a table.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Operations

```
/**
 *Obtain the names of all search indexes associated with the table.
 * @api
 *
 * @param [] $request
 *            The request parameter, which is the table name.
 * @return [] The response.
 * @throws OTSClientException The exception that is returned when a parameter error occurs
or the Tablestore server returns a verification error.
 * @throws OTSServerException The exception that is returned when the Tablestore server ret
urns an error.
 * @example "src/examples/ListSearchIndex.php"
 */
public function listSearchIndex(array $request)
```

Parameters

Parameter	Description

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

Examples

```
$request = array(
    'table_name' => 'php_sdk_test',
);
$response = $otsClient->listSearchIndex($request);
```

7.7.5. Term query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.

Parameter	Description
query_type	The query type. To use term query, set this parameter to QueryTypeConst::TERM_QUERY.
field_name	The name of the field used to match the query conditions.
term	The keyword used for the query to match the column values. The keyword is not tokenized but used as an exact word to match the column values. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that can match the keyword. For example, if the value of a TEXT field in a row is "tablestore is cool", the value can be tokenized into "tablestore", "is", and "cool". The row can meet query conditions when you set "tablestore", "is", or "cool" to the keyword to match the values of the TEXT field.
sort	The sorting method. The returned rows are sorted based on the specified method. For more information, see Sorting and pagination.
columns_to_get	Specifies whether to return all columns of each matched row. You can configure return_type and return_names for this parameter. • If you set return_type to ColumnReturnTypeConst::RETURN_SPECIFIED, you can use return_names to specify the columns to return. • If return_type is set to ColumnReturnTypeConst::RETURN_ALL, all columns are returned. • If you set return_type to ColumnReturnTypeConst::RETURN_NONE, only the primary key columns are returned.

Examples

```
$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::TERM_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'term' => 'keyword'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_ALL,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

7.7.6. Terms query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
offset	The position from which the current query starts.

Parameter	Description
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
query_type	The query type. To use terms query, set this parameter to QueryTypeConst::TERMS_QUERY.
field_name	The name of the field used to match the query conditions.
terms	The keywords used in the query to match the column values. A row meets the query conditions when the value of the specified column exactly matches at least one keyword.
sort	The sorting method. The returned rows are sorted based on the specified method. For more information, see Sorting and pagination.
columns_to_get	Specifies whether to return all columns of each matched row. You can configure return_type and return_names for this parameter. • If you set return_type to ColumnReturnTypeConst::RETURN_SPECIFIED, you can configure return_names to specify the columns to return. • If return_type is set to ColumnReturnTypeConst::RETURN_ALL, all columns are returned. • If you set return_type to ColumnReturnTypeConst::RETURN_NONE, only the primary key columns are returned.

Examples

```
$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' => QueryTypeConst::TERMS_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'terms' => array(
                    "keyword",
                    "php"
                )
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'long',
                    'order' => SortOrderConst::SORT_ORDER_DESC,
                    'mode' => SortModeConst::SORT_MODE_AVG
                )
            )
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

7.7.7. Match all query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the base table.

Parameter	Description
index_name	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
collapse	Specifies that the returned results are collapsed based on the specified column.
query_type	The query type. To use match query, set this parameter to QueryTypeConst::MATCH_ALL_QUERY.
sort	The sorting method. The returned rows are sorted based on the specified method. For more information, see Sorting and pagination.
columns_to_get	Specifies whether to return all columns of each matched row. • If you set return_type to ColumnReturnTypeConst::RETURN_SPECIFIED, you can use return_names to specify the columns to return. • If return_type is set to ColumnReturnTypeConst::RETURN_ALL, all columns are returned. • If you set return_type to ColumnReturnTypeConst::RETURN_NONE, only the primary key columns are returned.

Example

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'collapse' => array(
            'field_name' => 'keyword'
        ),
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
//        'sort' => array(// Set a specific sorting method if required.
//            array(
//                'field_sort' => array(
//                    'field_name' => 'keyword',
//                    'order' => SortOrderConst::SORT_ORDER_ASC
//                )
//            ),
//        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    )
);
$response = $otsClient->search($request);
```

7.7.8. Match query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.

Parameter	Description
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
query_type	The query type. To use match query, set this parameter to QueryTypeConst::MATCH_QUERY.
field_name	The name of the column used to match the query conditions. Match query applies to TEXT columns.
text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, you can specify a TEXT column for a query, and set the analyzer to perform single-word tokenization. If you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore", "is this tablestore", "tablestore is cool", "this", and "is".
operator	The logical operator. By default, OR is used as the logical operator, which indicates that a row matches the query conditions when one of the analyzed keywords is matched. If you set operator to AND, a row meets the query condition only when all tokenized keywords are matched.
minimum_should_match	The minimum number of matched tokenized keywords contained in a column value. A row is returned only when the value of the queried column in the row contains at least the minimum number of matched tokenized keywords. **Note** minimum_should_match must be used together with the OR logical operator.

Parameter	Description
columns_to_get	Specifies whether to return all columns of each matched rows. You can configure return_type and return_names for this parameter. • If you set return_type to ColumnReturnTypeConst::RETURN_SPECIFIED, you can use return_names to specify the columns to return. • If you set return_type to ColumnReturnTypeConst::RETURN_ALL, all columns are returned. • If you set return_type to ColumnReturnTypeConst::RETURN_NONE, only the primary key columns are returned.

Examples

```
$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::MATCH_QUERY,
            'query' => array(
                'field_name' => 'text',
                'text' => 'ots text php keyword',
//              'operator' => QueryOperatorConst::PBAND,
                'operator' => QueryOperatorConst::PBOR,// Use minimum_should_match with OR.
                'minimum_should_match' => 3
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('text')
    )
);
$response = $otsClient->search($request);
```

7.7.9. Phrase query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set Limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specify whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to true.
query_type	The query type. To use phrase query, set this parameter to QueryTypeConst::MATCH_PHRASE_QUERY.
field_name	The name of the column used to match the query conditions. Match query applies to TEXT columns.
text	The keyword used for the query to match the column values. If the column used for the match query is a TEXT column, the keyword is tokenized into multiple keywords based on the analyzer you set when you create the search index. By default, single-word tokenization is performed if you do not set the analyzer when you create the search index. For example, if you use the phrase "this is" as the keyword for a query, you can obtain query results such as "..., this is tablestore" and "this is a table". Query results such as "this table is ..." or "is this a table" cannot be obtained.

Parameter	Description
columns_to_get	Specifies whether to return all columns of each matched rows. • If you set return_type to ColumnReturnTypeConst::RETURN_SPECIFIED, you can use return_names to specify the columns to return. • If you set return_type to ColumnReturnTypeConst::RETURN_ALL, all columns are returned. • If you set return_type to ColumnReturnTypeConst::RETURN_NONE, only the primary key columns are returned.

Examples

```
$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::MATCH_PHRASE_QUERY,
            'query' => array(
                'field_name' => 'text',
                'text' => 'text keyword'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('text')
    )
);
$response = $otsClient->search($request);
```

7.7.10. Prefix query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.

- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to false, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
query_type	The query type. To use prefix query, set this parameter to QueryTypeConst::PREFIX_QUERY.
field_name	The name of the field used to match the query conditions.
prefix	The value of the prefix. If the field used to match the query conditions is a TEXT field, the field values are tokenized. A row meets the query conditions when the tokenized value of the specified field contains at least one term that contains the specified prefix.
sort	The sorting method. The returned rows are sorted based on the specified method. For more information, see Sorting and pagination.
columns_to_get	Specifies whether to return all columns of each matched row. You can configure return_type and return_names for this parameter. • If you set return_type to ColumnReturnTypeConst::RETURN_SPECIFIED, you can use return_names to specify the columns to return. • If return_type is set to ColumnReturnTypeConst::RETURN_ALL, all columns are returned. • If you set return_type to ColumnReturnTypeConst::RETURN_NONE, only the primary key columns are returned.

Examples

```
$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::PREFIX_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'prefix' => 'key'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_ALL,
        'return_names' => array('keyword')
    )
);
$response = $otsClient->search($request);
```

7.7.11. Range query

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
offset	The position from which the current query starts.

Parameter	Description
limit	The maximum number of rows that the current query returns. To query only the number of matched rows but not detailed data, you can set limit to 0. In this case, Tablestore returns the number of matched rows without data from the table.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, this parameter is set to False, which indicates that the total number of rows that match the query conditions is not returned. Query performance is affected when this parameter is set to True.
query_type	The query type. To use range query, set this parameter to QueryTypeConst::RANGE_QUERY.
field_name	The name of the field used to match the query conditions.
range_from	The value from which the query starts.
range_to	The value with which the query ends.
include_lower	Specifies whether to include the value of range_from in the response. The value type is Boolean.
include_upper	Specifies whether to include the value of range_to in the response. The value type is Boolean.
sort	The sorting method. The returned rows are sorted based on the specified method. For more information, see Sorting and pagination.
columns_to_get	Specifies whether to return all columns of each matched row. You can configure return_type and column_names for this parameter. • If you set return_type to ColumnReturnType.SPECIFIED, you can configure column_names to specify the columns to return. • If you set return_type to ColumnReturnType.ALL, all columns are returned. • If you set return_type to ColumnReturnType.NONE, only the primary key columns are returned.

Examples

```
$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::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' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('double', 'long', 'keyword')
    )
);
$response = $otsClient->search($request);
```

7.7.12. Wildcard query

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

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the data table.
index_name	The name of the search index.
offset	The position from which the current query starts.
limit	The maximum number of rows that you want the current query to return.
get_total_count	Specifies whether to return the total number of rows that match the query conditions. By default, the value of this parameter is false, which specifies that the total number of rows that match the query conditions is not returned. If you set the get_total_count parameter to true, the query performance is compromised.
query_type	The query type. Set the query type to QueryTypeConst::WILDCARD_QUERY.
field_name	The name of the column.
value	The string that contains wildcards. The string cannot exceed 32 characters in length.
sort	The sorting method. The returned rows are sorted based on the method that you specify. For more information, see Sorting and pagination.
columns_to_get	Specifies whether to return all columns of each matched row. • If you set the return_type parameter to ColumnReturnTypeConst::RETURN_SPECIFIED, you must use return_names to specify the columns that you want to return. • If you set the return_type parameter to ColumnReturnTypeConst::RETURN_ALL, all columns are returned. • If you set the return_type parameter to ColumnReturnTypeConst::RETURN_NONE, only the primary key columns are returned.

Examples

```
$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::WILDCARD_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'value' => 'key*'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword')
    )
);
$response = $otsClient->search($request);
```

7.7.13. Geo query

Geo query includes geo-distance query, geo-bounding box query, and geo-polygon query.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Geo-distance query

You can use geo-distance query to specify a circular geographical area consisting of a central point and radius as a filter condition in a query. Tablestore returns the rows where the values of a specified column fall within the circular geographical area.

- Parameters

Parameter	Description
table_name	The name of the table.

Parameter	Description
index_name	The name of the search index.
query	The query statement for the search index. To use geo-distance query, set the query type to QueryTypeConst::GEO_DISTANCE_QUERY.
field_name	The name of the column used to match the query conditions. The specified column must be a GEOPOINT column.
center_point	The coordinate pair of the central point. The value of this parameter is of the GEOPOINT data type in the following format: `latitude,longitude`. Valid values of the latitude: [-90,+90]. Valid values of the longitude: [-180,+180]. Example: `35.8,-45.91`.
distance	The radius of the circular geographical area. The value of this parameter is of the DOUBLE data type. Unit: meters.

- Examples

  The following code provides an example on how to query rows where the values of the geo column fall within the circular geographical area whose central point is `30.001,120.001` and radius is 1,000 meters:

```
$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_QUERY,
            'query' => array(
                'field_name' => 'geo',
                'center_point' => '30.001,120.001',
                'distance' => 1000
            )
        ),
        'sort' => array(
            array(
                'geo_distance_sort' => array(
                    'field_name' => 'geo',
                    'order' => SortOrderConst::SORT_ORDER_ASC,
                    'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
                    'points' => array('30,120')
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('geo')
    )
);
$response = $otsClient->search($request);
```

Geo-bounding box query

You can use geo-distance query to specify a rectangular geographical area. Tablestore returns the rows where the values of a specified column fall within the rectangular geographical area.

- Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
query	The query statement for the search index. To use geo-bounding query, set the query type to QueryTypeConst::GEO_BOUNDING_BOX_QUERY.
field_name	The name of the column used to match the query conditions. The specified column must be a GEOPOINT column.
top_left	The coordinate pair of the upper-left corner of the rectangle box.

Parameter	Description
bottom_right	The coordinate pair of the lower-right corner of the rectangle box. A rectangular geographical area can be specified by a upper-left corner and a lower-right corner. The value of this parameter is in the following format: `latitude,longitude`. Valid values of the latitude: [-90,+90]. Valid values of the longitude: [-180,+180]. Example: `35.8,-45.91`.

- Examples

  The following code provides an example on how to query rows where the values of the geo column fall within the rectangular geographical area whose upper-left corner is at `31,119` and lower-right corner at `29,121`.

```
$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_BOUNDING_BOX_QUERY,
            'query' => array(
                'field_name' => 'geo',
                'top_left' => '31,119',
                'bottom_right' => '29,121'
            )
        ),
        'sort' => array(
            array(
                'geo_distance_sort' => array(
                    'field_name' => 'geo',
                    'order' => SortOrderConst::SORT_ORDER_ASC,
                    'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
                    'points' => array('30,120')
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('geo')
    )
);
$response = $otsClient->search($request);
```

Geo-polygon query

You can use geo-polygon query to specify a polygonal geographical area. Tablestore returns the rows where the values of a specified column fall within the polygonal geographical area.

- Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
query	The query statement for the search index. To use geo-polygon query, set the query type to QueryTypeConst::GEO_POLYGON_QUERY.
field_name	The name of the column used to match the query conditions. The specified column must be a GEOPOINT column.
points	The coordinate pairs of points that uniquely define the polygon geographical area. You can specify a polygon by using multiple coordinates in the following format: `latitude,longitude`. Valid values of the latitude: [-90,+90]. Valid values of the longitude: [-180,+180]. Example: `35.8,-45.91`.

- Examples

  The following code provides an example on how to query rows where the values of the geo column fall within the polygonal geographical area defined by the coordinate pairs of `31,120`, `29,121`, and `29,119`:

```
$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_POLYGON_QUERY,
            'query' => array(
                'field_name' => 'geo',
                'points' => array(
                    "31,120",
                    "29,121",
                    "29,119"
                )
            )
        ),
        'sort' => array(
            array(
                'geo_distance_sort' => array(
                    'field_name' => 'geo',
                    'order' => SortOrderConst::SORT_ORDER_ASC,
                    'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
                    'points' => array('30,120')
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('geo')
    )
);
$response = $otsClient->search($request);
```

7.7.14. Exists query

Exists query is also called NULL query or NULL-value query. This query is used in sparse data to determine whether a column of a row exists. For example, you can query the rows in which the value of the address column is not empty.

> **Note**
>
> - If you want to check whether a column contains empty values, you must use ExistsQuery together with must_not_queries of BoolQuery.
> - If one of the following conditions is met, the system considers that a column does not to exist. In this example, the city column is used.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the city column does not exist in the data table, the search index considers that the city column does not exist.
>   - The type of the city column in the search index is a basic type such as keyword. If a row in which the value of the city column is an empty array in the data table ("city" = "[]"), the search index considers that the city column does not exist.

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the data table.
index_name	The name of the search index.
field_name	The name of the column.
query_type	The query type. Set the query type to QueryTypeConst::EXISTS_QUERY. **Note** You cannot directly query nested columns. To query a nested column, you must encapsulate NestedQuery by specifying the path of the nested column and a subquery. The subquery can be a query of any type. For more information, see Nested query.

Examples

- Example 1

  The following sample code shows how to query the rows in which the text column is not empty:

```
$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::EXISTS_QUERY,
            'query' => array(
                'field_name' => 'text'
            )
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long', 'array')
    )
);
$response = $otsClient->search($request);
```

- Example 2

  The following sample code shows how to query the rows in which the nested_long subcolumn of the nested column is not empty:

```
$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::NESTED_QUERY,
            'query' => array(
                'path' => "nested",
                'query' => array(
                    'query_type' => QueryTypeConst::EXISTS_QUERY,
                    'query' => array(
                        'field_name' => 'nested.nested_long',
                    )
                ),
                'score_mode' => ScoreModeConst::SCORE_MODE_AVG
            )
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('nested')
    )
);
$response = $this->otsClient->search($request);
```

7.7.15. Collapse (distinct)

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
query	The query type. You can set this parameter to any query type.
collapse	The configuration of the collapse parameters, including field_name. field_name: the name of the column based on which the result set is collapsed. Only columns whose values are of the INTEGER, FLOATING-POINT and KEYWORD data types are supported.
offset	The position from which the current query starts.
limit	The maximum number of rows that the current query returns. To query only the number of matched rows without returning specific data, you can set limit to 0. Then, Tablestore returns the number of matched rows without data from the table.

Example

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'collapse' => array(
            'field_name' => 'keyword'
        ),
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
//        'sort' => array(// Set a specific sorting method if required.
//            array(
//                'field_sort' => array(
//                    'field_name' => 'keyword',
//                    'order' => SortOrderConst::SORT_ORDER_ASC
//                )
//            ),
//        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    )
);
$response = $otsClient->search($request);
```

7.7.16. Boolean query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.
- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the data table.
index_name	The name of the search index.

Parameter	Description
must_queries	The list of subqueries that the query results must match. This parameter is equivalent to the AND operator.
must_not_queries	The list of subqueries that the query results must not match. This parameter is equivalent to the NOT operator.
filter_queries	The list of subqueries. Only rows that match all subfilters are returned. filter is similar to query except that filter does not calculate the relevance score based on the number of subfilters that the row matches.
should_queries	The list of subqueries that the query results can or cannot match. This parameter is equivalent to the OR operator. Only rows that meet the minimum number of subquery conditions specified by should_queries are returned. A higher overall relevance score indicates that more subquery conditions specified by should_queries are met.
minimum_should_match	The minimum number of subquery conditions specified by should_queries that the rows must meet.If no other subquery conditions except the subquery conditions that are specified by should_queries are specified, the default value of the minimum_should_match parameter is 1. If other subquery conditions, such as subquery conditions specified by must_queries, must_not_queries, and filter_queries, are specified, the default value of the minimum_should_match parameter is 0.

Examples

In the following example, condition A requires that the value of the keyword column exactly matches keyword, and condition B requires that the value of the long column is greater than or equal to 100 and less than 101.

- Example 1

  The following sample code provides an example on how to perform a Boolean query to query rows that meet condition A and condition B:

```
$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' => QueryTypeConst::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' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- Example 2

  The following sample code provides an example on how to perform a Boolean query to query rows that do not meet both condition A and condition B:

```
$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_not_queries' => array(
                    array(
                        'query_type' => QueryTypeConst::TERM_QUERY,
                        'query' => array(
                            'field_name' => 'keyword',
                            'term' => 'keyword'
                        )
                    ),
                    array(
                        'query_type' => QueryTypeConst::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' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- Example 3

  The following sample code provides an example on how to perform a Boolean query to filter the rows and return rows that meet condition A and condition B. filter does not calculate the relevance score based on the number of subfilters that the rows match.

```
$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(
                'filter_queries' => array(
                    array(
                        'query_type' => QueryTypeConst::TERM_QUERY,
                        'query' => array(
                            'field_name' => 'keyword',
                            'term' => 'keyword'
                        )
                    ),
                    array(
                        'query_type' => QueryTypeConst::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' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- ShouldQueries(A OR B)

  The following sample code provides an example on how to query rows that meet at least one of condition A and condition B:

```
$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(
                'should_queries' => array(
                    array(
                        'query_type' => QueryTypeConst::TERM_QUERY,
                        'query' => array(
                            'field_name' => 'keyword',
                            'term' => 'keyword'
                        )
                    ),
                    array(
                        'query_type' => QueryTypeConst::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' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- Example 4

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

```
$request = array(
```

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_index',
    'search_query' => [
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => false,
        'query' => [
            'query_type' => QueryTypeConst::BOOL_QUERY,
            'query' => [
                // Final combination: (col2 < 4 or col3 < 5) or (col2 = 4 and (col3 = 5 o
r col3 = 6))
                'should_queries' => [
                    [
                        'query_type' => QueryTypeConst::BOOL_QUERY,
                        'query' => [
                            // Combination 1: col2 < 4 or col3 < 5
                            'should_queries' => [
                                [
                                    'query_type' => QueryTypeConst::RANGE_QUERY,
                                    // Condition 1: col2 < 4
                                    'query' => [
                                        'field_name' => 'col2',
                                        'range_to' => 4
                                    ]
                                ],
                                [
                                    'query_type' => QueryTypeConst::RANGE_QUERY,
                                    // Condition 2: col3 < 5
                                    'query' => [
                                        'field_name' => 'col3',
                                        'range_to' => 5
                                    ]
                                ]
                            ]
                        ]
                    ],
                    [
                        'query_type' => QueryTypeConst::BOOL_QUERY,
                        // Combination 2: (col2 = 4 and (col3 = 5 or col3 = 6))
                        'query' => [
                            'must_queries' => [
                                [
                                    'query_type' => QueryTypeConst::TERM_QUERY,
                                    // Condition 3: col2 = 4
                                    'query' => [
                                        'field_name' => 'col2',
                                        'term' => 4
                                    ]
                                ],
                                [
                                    'query_type' => QueryTypeConst::BOOL_QUERY,
                                    // Combination 3: (col3 = 5 or col3 = 6)
                                    'query' => [
                                        'should_queries' => [
```

```
                                            [
                                                'query_type' => QueryTypeConst::TERM_QUER
Y,
                                                // Condition 4: col3 = 5
                                                'query' => [
                                                    'field_name' => 'col3',
                                                    'term' => 5
                                                ]
                                            ],
                                            [
                                                'query_type' => QueryTypeConst::TERM_QUER
Y,
                                                // Condition 5: col3 = 6
                                                'query' => [
                                                    'field_name' => 'col3',
                                                    'term' => 6
                                                ]
                                            ]
                                        ],
                                        'minimum_should_match' => 1
                                    ]
                                ]
                            ]
                        ]
                    ],
                    'minimum_should_match' => 1
                ]
            ]
        ],
        // You can configure the columns_to_get parameter to specify the columns to return or
specify that all columns are returned. If you do not configure this parameter, only the p
rimary key columns are returned.
        'columns_to_get' => [
            //'return_type' => ColumnReturnTypeConst::RETURN_ALL // Specify that all columns
are returned.
            // Specify that the col1 and col2 columns are returned.
            'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
            'return_names' => array('col1', 'col2')
        ]
);
```

7.7.17. Nested query

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

Prerequisites

- The OTSClient is initialized. For more information, see Initialization.
- A data table is created. Data is written to the table.

- A search index is created for the data table. For more information, see Create search indexes.

Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the search index.
path	The path of the subcolumn in the nested column. The path is similar to the tree structure. For example, news.title indicates the title subcolumn in the nested news column.
query	The query on the subcolumn in the nested column. The query can be of any query type.
score_mode	Specifies which value is used to calculate the score when a column contains multiple values.

Examples

The following code provides an example on how to search the nested.nested_keyword column for data that matches "sub":

```
$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::NESTED_QUERY,
            'score_mode' => ScoreModeConst::SCORE_MODE_AVG,
            'query' => array(
                'path' => "nested",
                'query' => array(
                    'query_type' => QueryTypeConst::TERM_QUERY,
                    'query' => array(
                        'field_name' => 'nested.nested_keyword',
                        'term' => 'sub'
                    )
                )
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'nested.nested_long',
                    'order' => SortOrderConst::SORT_ORDER_DESC,
                    'nested_filter' => array(
                        'path' => "nested",
                        'query' => array(
                            'query_type' => QueryTypeConst::TERM_QUERY,
                            'query' => array(
                                'field_name' => 'nested.nested_keyword',
                                'term' => 'sub'
                            )
                        )
                    )
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('nested')
    )
);
$response = $otsClient->search($request);
```

7.7.18. Sorting and pagination

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

Index presorting

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

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

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

Specify a sorting method

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

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

- ScoreSort

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

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

  ```
  'sort' => array(
      array(
          'score_sort' => array(
              'order' => SortOrderConst::SORT_ORDER_DESC
          )
      ),
  )
  ```

- PrimaryKeySort

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

  ```
  'sort' => array(
      array(
          'pk_sort' => array(
              'order' => SortOrderConst::SORT_ORDER_ASC
          )
      ),
  )
  ```

- FieldSort

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

```
'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 results by geographical location.

```
'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 based on the order of multiple columns.

```
'sort' => array(
    array(
        'field_sort' => array(
            'field_name' => 'keyword',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
    array(
        'pk_sort' => array(
            'order' => SortOrderConst::SORT_ORDER_ASC
        )
    ),
)
```

Specify a pagination method

You can use limit and offset or use tokens to split returned query results into pages.

- Use the limit and offset parameters

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

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

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

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 10,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    )
);
$response = $otsClient->search($request);
```

- Use a token

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

  If Tablestore is not able to complete reading data that meets the filter condition, the server returns next_token. You can use next_token to continue reading the subsequent data.

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

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

Notice Search indexes of the nested type do not support IndexSort. If you use a search index of the nested type to query data and require pagination, you must specify the sorting method to return data in the query conditions. Otherwise, Tablestore does not return next_token when all data that meets the query conditions is not completely read.

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::FUNCTION_SCORE_QUERY,
            'query' => array(
                'query' => array(
                    'query_type' => QueryTypeConst::TERM_QUERY,
                    'query' => array(
                        'field_name' => 'keyword',
                        'term' => 'keyword'
                    )
                ),
                'field_value_factor' => array(
                    'field_name' => 'long'
                )
            )
        ),
        'sort' => array(
            array(
                'score_sort' => array(
                    'order' => SortOrderConst::SORT_ORDER_DESC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::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);
}
```

7.8. Global secondary index

After you create an index table for a data table, you can read data from the index table or delete the specified index table.

Prerequisites

- The OTSClient instance is initialized. For more information, see Initialization.
- A table is created. The value of timeToLive is -1. The value of maxVersions is 1.
- Predefined columns are set for the table.

Create an index table by calling the CreateIndex operation

You can call the CreateIndex operation to create an index table for an existing base table.

> **Note** You can also create one or more index tables when you create a base table by calling the CreateTable operation. For more information, see Create tables.

- API operations

```
    /**
     * 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 returned when a parameter error o
ccurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore serv
er returns an error.
     * @example "src/examples/CreateIndex.php"
     */
    public function createIndex(array $request)
```

- Parameters

Parameter	Description
main_table_name	The name of the table.

Parameter	Description
index_meta	The schema information of the index table, which includes the following items: ○ name: the name of the index table. ○ primary_key: the primary key of the index table, which is a combination of primary key columns and predefined columns of the base table. ○ defined_column: the indexed attribute column, which is a combination of predefined columns of the base table. ○ index_update_mode: the update mode of the index table. Only IUM_ASYNC_INDEX is supported. ○ index_type: the type of the index table. Only IT_GLOBAL_INDEX is supported.
include_base_data	Specifies whether the index table includes the existing data in the base table. When the include_base_data parameter in CreateIndex is set to true, the index includes the existing data. When the value is false, the index excludes the existing data.

- Examples

```
$request = array(
    'table_name' => 'MainTableName',
    'index_meta' => array(
        'name' => self::$indexName1,
        'primary_key' => array('col1'),
        'defined_column' => array('col2')
    )
);
$this->otsClient->createIndex($request);
```

Read data from the index table

You can read a row of data or read data within a specified range from the index table. If the columns to return are included in the index table, you can query the data from the index table. Otherwise, you must query the base table for the required data.

- Read a row of data from the index table

  For more information, see Single-row operations.

  When you use the GetRow operation to read data from the index table, take note of the following items:

  ○ You must set table_name to the name of the index table.
  ○ Tablestore autocompletes the primary key that is not configured in the index columns for the index table. Therefore, when you specify the primary key of a row, you must specify the indexed columns based on which you create the index table and the autocompleted primary key columns.

- Read data within a specified range from the index table

For more information, see Multi-row operations.

When you use the GetRange operation to read data from the index table, take note of the following items:

- You must set table_name to the name of the index table.
- Tablestore autocompletes the primary key that is not configured in the index columns for the index table. Therefore, when you specify the start primary key and the end primary key of the range, you must specify the indexed columns based on which you create the index table and the autocompleted primary key columns.

Delete a index table (DropIndex)

You can call the DropIndex operation to delete a specified index table from a base table.

- API operations

```
    /**
     * 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 returned when a parameter error o
ccurs or the Tablestore server returns a verification error.
     * @throws OTSServerException The exception that is returned when the Tablestore serv
er returns an error.
     * @example "src/examples/DropIndex.php"
     */
    public function dropIndex(array $request)
```

- Parameters

Parameter	Description
table_name	The name of the table.
index_name	The name of the index table.

- Examples

```
$request = array (
    'table_name' => 'MainTableName',
    'index_name' => 'IndexName'
);
$otsClient->dropIndex( $request );
```

7.9. Troubleshooting

This topic describes the troubleshooting methods in Tablestore SDK for PHP.

Methods

Tablestore SDK for PHP handles errors as exceptions. The operation is successful if no exceptions are returned for the called operation. If an exception is returned, the operation fails.

> **Note** Batch operations such as BatchGetRow and BatchWriteRow can be called only after the system verifies that no exception is returned and that the status of each row is successful.

Exceptions

Tablestore SDK for PHP has two types of exceptions: OTSClientException and OTSServerException. Both are inherited from the Exception class.

- OTSClientException: an internal SDK exception such as an invalid parameter value.
- OTSServerException: indicates a server error that arises from the resolution of a server error message. OTSServerException has the following components:
  - getHttpStatus(): an HTTP status code such as 200 or 404.
  - getOTSErrorCode(): the error type string returned by Tablestore.
  - getOTSErrorMessage(): the detailed error description returned by Tablestore.
  - getRequestId(): the UUID that identifies a request. If a problem persists, record this request ID and submit a ticket.

8.Download SDK

8.1. Tablestore SDK for Java

This topic describes the version history of Tablestore SDK for Java.

> **Note** Tablestore SDK for Java V4.0.0 and later support max versions and time to live (TTL), and are incompatible with Tablestore SDK for Java V2.x.x.

Version: V5.13.5

Release date: 2022-01-06

Download link: 5.13.5

Change records:

- Time series metadata in the TimeSeries model can be deleted. For more information, see Delete time series metadata.
- Queried time series data in the TimeSeries model can be sorted in reverse chronological order. For more information, see Query time series data.

Version: V5.13.0

Release date: 2021-11-17

Download link: 5.13.0

Change records: The SQL query feature is added. For more information, see Overview.

Version: V5.12.0

Release date: 2021-10-27

Download link: 5.12.0

Change records: The TimeSeries model is added. For more information, see Overview.

Version: V5.11.0

Release date: 2021-07-23

Download link: 5.11.0

Change records:

- The virtual column feature is added to search indexes. For more information, see Virtual columns.
- Aggregation by percentile and aggregation by histogram are added to search indexes. For more information, see Aggregation.

Version: V5.10.3

Release date: 2020-10-26

Download link: 5.10.3

Change records:

- The data delivery feature is added. For more information about how to use the Tablestore console to deliver data, see Quick start. For more information about how to use Tablestore SDKs to deliver data, see Use SDKs.
- A row query feature is supported for search indexes. You can use this feature to query the rows that are obtained from the results of an aggregation operation in each group. For more information, see Aggregation.
- The format check feature for the configuration of parameters is added.
- The connection to compute engines is optimized for Tablestore SDK for Java.

Version: V5.7.0

Release date: 2020-04-07

Download link: 5.7.0

Change records:

- Parallel scan is added to search indexes. For more information, see Parallel scan.
- TableStoreWriter is optimized.

Version: V5.4.0

Release date: 2019-11-18

Download link: 5.4.0

Change records:

- Predefined columns can be added to or deleted from existing tables.
- Tunnel Service is optimized for Tablestore SDK for Java.

Version: V5.3.0

Release date: 2019-09-10

Download link: 5.3.0

Change records: Aggregation operations can be performed on data when search indexes are used.

Version: V5.1.0

Release date: 2019-07-01

Download link: 5.1.0

Change records: More tokenization methods are added to search indexes.

Version: V5.0.0

Release date: 2019-06-10

Download link: 5.0.0

Change records:

- To unify SDK version numbers, Tablestore SDK for Java is updated to V5.x. Tablestore SDK for Java V5.x is compatible with Tablestore SDK for Java V4.x.
- Exists query is added to search indexes.

Version: V4.12.1

Release date: 2019-05-08

Download link: 4.12.1

Change records: Timeline model V2 is added.

Version: V4.10.2

Release date: 2019-03-11

Download link: tablestore-4.10.2

Change records:

- Remove log4j2 implement and log4j2.xml.
- TunnelWorker: fix auto retry logic when network condition is too bad.

Version: V4.7.4

Release date: 2018-09-27

Download link: tablestore-4.7.4-release.zip

Change records:

- Search indexes are supported, and the following query types are added to search indexes:
  - Boolean query
  - Range query
  - Wildcard query
  - Nested query
  - Full-text query
  - Sorting
- The global secondary index feature is added.

Version: V4.0.0

Release date: 2016-08-01

Download link: aliyun_tablestore_java_sdk_4.0.0.zip

Change records:

- TTL is added. For more information, see the "TTL" section in Max versions and TTL.
- The max versions feature is added. For more information, see the "Max versions" section in Max versions and TTL.

Version: V2.2.4

Release date: 2016-05-12

Download link: aliyun_tablestore_java_sdk_2.2.4.zip

Change records:

- The conditional update feature is added.
- The filter feature is added.

Version: V2.1.0

Release date: 2015-11-12

Change records:

- Asynchronous network transmission and performance tuning: The queries per second (QPS) performance is increased by several times when CPU utilization remains unchanged.
- Flexible and simple asynchronous operations: Callbacks can be imported, and future can be returned at the same time.
- Content related to Object Storage Service (OSS) SDK removed: The version of Tablestore SDK for Java contains code that is related only to Tablestore SDK. The directory structure is fine-tuned.
- Retry logic optimization: The default retry logic is optimized. Retry attempts are supported for single-row errors that occur when batch operations are performed. A simpler method is provided to allow you to modify retry logic.
- Log optimization: Detailed logs are recorded for the time period from when a request is sent to when the request is received. Log records of slow requests are provided. Trace IDs can be used to view the comprehensive log records for the requests that are sent between Tablestore SDK for Java and the backend service.
- OTSWriter that can be used to import a large amount of data at a time: provides a simple and efficient data import experience.
- Other improvements: Tool functions that support multiple data types are provided. The operation that can be called to calculate data sizes is provided.

Note:

The following compatibility issues exist between Tablestore SDK for Java V2.1.0 and V2.0.4:

- The import path for specific classes must be modified after the later version of Tablestore SDK for Java is used. Packages of specific classes are modified. For example, the package of ClientConfiguration is changed from com.aliyun.openservices to com.aliyun.openservices.ots. These packages are modified because Tablestore SDK is separated from OSS SDK. The classes shared by both SDKs must be included in the packages of Tablestore.
- If you no longer use an OTSClient instance, for example, before the program exits, you must call the shutdown method of OTSClient to release all threads and connection resources of the OTSClient instance.
- The names of specific configuration items in ClientConfiguration are changed. For example, time units are added to the names.
- Modifications are made to dependencies of Tablestore SDK for Java V2.1.0. For example, HttpAsyncClient and Joda-Time are used. If errors occur when you run the program, check whether the imported dependencies are the same as the existing dependencies.

Version: V2.0.4

Release date: 2015-09-25

8.2. Tablestore SDK for Node.js

This topic describes the version history of Tablestore SDK for Node.js.

Version: 4.0.6

Release date: October 9, 2016

Download URL: aliyun-tablestore-nodejs-sdk-4.0.6.tar.gz

GitHub: v4.0.6

Change records:

- The new feature async await is supported, and the corresponding sample code are added.
- The problem that part of Chinese characters are incorrectly read is fixed.
- STSToken is supported.
- The maxTimeDeviation condition is supported by the UpdateTable operation.
- The logos for version, build, coverage, license are added to the README document.

Version: 4.0.3

Release date: August 27, 2016

Download URL: aliyun-tablestore-nodejs-sdk-4.0.3.tar.gz

GitHub: v4.0.3

Change records:

- The version information is unified.

Version: 4.0.1

Release date: August 27, 2017

Download URL: aliyun-tablestore-nodejs-sdk-4.0.1.tar.gz

GitHub: v4.0.1

Change records:

- Part of invalid code is removed.
- The problem in response structure that occurs when no data is returned for the GetRange request is fixed.

Version: 4.0.0

Release date: July 7, 2017

Download URL: aliyun-tablestore-nodejs-sdk-4.0.0.tar.gz

GitHub: v4.0.0

8.3. Tablestore SDK for Python

This topic describes the version history of Tablestore SDK for Python.

Version: 5.2.1

Release date: February 20, 2021

Download URL: aliyun-tablestore-python-sdk-5.2.1.tar.gz

Change records: Aggregation can be performed on data when search index is used. For more information, see Aggregation.

Version: 5.1.0

Release date: July 22, 2019

Download URL: aliyun-tablestore-python-sdk-5.1.0.tar.gz

Change records: Local transactions are supported. For more information, see Configure local transaction.

Version: 5.0.0

Release date: May 29, 2019

Download URL: aliyun-tablestore-python-sdk-5.0.0.tar.gz

Change records:

- Column existence queries are supported in search index.
- The problems that occur when next_token does not exist are fixed.

Version: 4.6.0

Release date: February 15, 2019

Download URL: aliyun-tablestore-python-sdk-4.6.0.tar.gz

Change records:

- Search index is supported.
- Global secondary index is supported.

Version: 4.3.0

Release date: December 12, 2017

Download URL: aliyun-tablestore-python-sdk-4.3.0.tar.gz

Change records: Tablestore SDK for Python is compatible with Python 3, including Python 3.3, Python 3.4, Python 3.5, and Python 3.6.

Version: 4.2.0

Release date: September 12, 2017

Download URL: aliyun-tablestore-python-sdk-4.2.0.tar.gz

Change records: STS is supported.

Version: 4.1.0

Release date: June 28, 2017

Download URL: aliyun-tablestore-python-sdk-4.1.0.tar.gz

Change records: Tablestore SDK for Python is compatible with Python 2.6.

Version: 4.0.0

Release date: June 27, 2017

Download URL: aliyun-tablestore-python-sdk-4.0.0.tar.gz

Change records:

- Max versions is supported. For more information, see Data versions and TTL.
- TTL is supported. For more information, see Data versions and TTL.

- Auto-increment of primary key columns is supported. For more information, see Configure an auto-increment primary key column.

Version: 2.1.0

Release date: October 15, 2016

Download URL: aliyun-ots-python-sdk-2.1.0.zip

Change records:

- Conditional update is supported. For more information, see Configure conditional update.
- Filters are supported. For more information, see Filter.
- The bug of invalid retry policies are fixed inside Tablestore SDK for Python.

Version: 2.0.8

Release date: March 30, 2016

Change records: Parameters that can be configured for the connection pool are adjusted to support access over HTTPS and certificate-based authentication.

Version: 2.0.7

Release date: December 30, 2015

Change records: Reserved capacity unit (CU) settings in the code example are modified based on pay-as-you-go.

Version: 2.0.6

Release date: October 23, 2015

Download URL: aliyun-ots-python-sdk-2.0.6.zip

Change records: The exponential backoff strategy is adjusted to process part of exceptions.

Version: 2.0.5

Release date: September 25, 2015

Download URL: ots_python_sdk-2.0.5.zip

8.4. Tablestore SDK for .NET

This topic describes the version history of Tablestore SDK for .NET.

Version: 4.1.4

Release date: June 24, 2019

Download URL: 4.1.4

Change records: The problem that single-word tokenization is performed on the keyword by default in search index is fixed.

Version: 4.1.2

Release date: May 29, 2019

Download URL: 4.1.2

Change records: The problem that tokenization can be performed on the keyword when the column used for the match query is not a TEXT column is fixed.

Version: 4.1.1

Release date: March 6, 2019

Download URL: 4.1.1

Change records: The problem that garbled characters are displayed in some scenarios in which search index is used is fixed.

Version: 4.1.0

Release date: December 17, 2018

Download URL: 4.1.0

Change records:

- Atomic counter is supported. For more information, see Configure atomic counter.
- Global secondary index is supported. For more information, see Global secondary index.
- Search index is supported. For more information, see Create search indexes.

Version: 4.0.0

Release date: September 18, 2018

Download URL: 4.0.0

Change records:

- Max versions and TTL are supported. For more information, see Data versions and TTL.
- Auto-increment primary key columns are supported. For more information, see Configure an auto-increment primary key column.

Version: 3.0.0

Release date: May 5, 2016

Change records:

- The filter feature is supported.
- The warnings generated during the compilation process are cleared.
- Unnecessary dependent packages and code are deleted.
- The code used to call templates is simplified.
- The check for invalid parameters is added.
- Spaces are removed from both ends of user-defined parameters by using the trim() function.
- The UserAgent header is added.
- The Condition.IGNORE, Condition, EXPECT_EXIST, and Condition.EXPECT_NOT_EXIST parameters are deleted.
- The DLL file name is changed from Aliyun.dll to Aliyun.TableStore.dll.
- The open source code is released to GitHub.

Version: 2.2.1

Release date: April 14, 2016

Change records: Response headers are case-insensitive when they are verified by the SDK.

Version: 2.2.0

Release date: April 5, 2016

Change records:

- The default number of connections in a connection pool is increased from 50 to 300.
- Conditional update is supported.

Version: 2.1.0

Release date: December 30, 2015

Change records:

- Reserved capacity unit (CU) settings in the code example are modified based on the pay-as-you-go billing method.
- More test cases for BatchGetRow and BatchWriteRow are provided.

8.5. 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:

Updates:

- Removes delete table from example code.

PHP SDK version 2.0.2

Release date: 11/04/2016

Updates:

- Fix a bug of pb decode.

PHP SDK version 2.0.0

Release date: 22/09/2015

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.