CloudTable Service

Developer Guide

Issue 01

Date 2024-06-20

HUAWEI TECHNOLOGIES CO., LTD.

Huawei Technologies Co., Ltd.

Address:	Huawei Industrial Base Bantian, Longgang Shenzhen 518129 People's Republic of China
Website:	https://www.huawei.com
Email:	support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 HBase Application Development Guide

1.1 Application Development Process

This section describes how to call open source APIs of HBase in the CloudTable cluster mode to develop Java applications.

Figure 1-1 and **Table 1-1** describe the phases in the development process.

Figure 1-1 Application development process

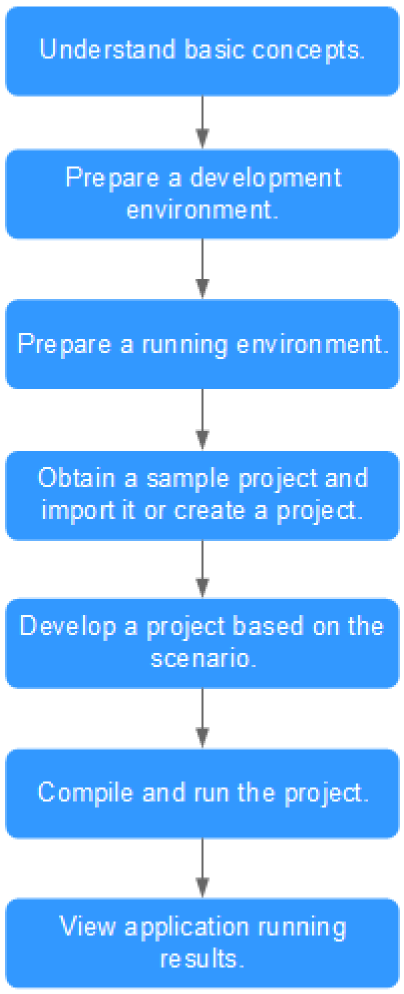

Understand basic concepts.
Prepare a development environment.
Prepare a running environment.
Obtain a sample project and import it or create a project.
Develop a project based on the scenario.
Compile and run the project.
View application running results.

Table 1-1 Application development process details

Phase	Description	Reference
Understand basic concepts.	Before application development, learn basic concepts of HBase, understand the scenario requirements, and design tables.	**HBase**
Prepare a development environment.	The Java language is recommended for HBase application development. You can use the Eclipse tool.	**Development Environment Introduction**
Prepare a running environment.	The application running environment is a client. Install and configure the client according to the guide.	**Preparing a Windows Running Environment**
Prepare a project.	CloudTable provides example projects for different scenarios. You can import an example project to learn the application. You can also create a project according to the guide.	**Downloading a Sample Project** **Configuring and Importing a Project**

Phase	Description	Reference
Develop a project based on the scenario.	An example project using Java is provided, including creating a table, writing data into the table, and deleting the table.	**Developing HBase Applications**
Compile and run the application.	You can compile the developed application and submit it for running.	**Compiling and Running Applications** **Compiling and Running an Application When a Client Is Installed** or **Compiling and Running an Application When No Client Is Installed**
View application running results.	Application running results are exported to a path you specify. You can also view the application running status on the UI.	• In Windows: **Viewing Commissioning Results** • In Linux: **Viewing Commissioning Results**

1.2 Preparing a Development Environment

1.2.1 Development Environment Introduction

Table 1-2 describes the environment required for secondary development.

Table 1-2 Development environment

Item	Description
OS	Windows OS. Windows 7 or later is recommended.

Item	Description
JDK installation	Basic configurations of the development environment. JDK 1.7 or 1.8 is required. You are recommended to use JDK 1.8 for better compatibility of later versions. **NOTE** For security purpose, CloudTable supports only TLS 1.1 and TLS 1.2 encryption protocols. IBM JDK supports only 1.0 by default. If you use IBM JDK, set **com.ibm.jsse2.overrideDefaultTLS** to **true**. After the parameter setting, TLS1.0/1.1/1.2 can be supported at the same time. For details, see the related instructions on the IBM official website.
Eclipse installation and configuration	It is a tool used to develop CloudTable applications.
Network	Ensure that the development environment or client can communicate with the network of the CloudTable server.

1.2.2 Preparing a Running Environment

1.2.2.1 Preparing a Windows Running Environment

Scenario

The running environment for CloudTable application development can be deployed on Windows. You can perform the following operations to prepare the running environment.

Procedure

Step 1 Check whether a CloudTable cluster is installed and run properly.

Step 2 Prepare a Windows ECS.

For details about how to prepare an ECS, see **Preparing an ECS**.

Step 3 Install JDK 1.7 or later on the Windows ECS. However, you are recommended to use JDK 1.8 or later and install Eclipse that uses JDK 1.7 or later.

NOTE

- If you use IBM JDK, ensure that the JDK configured in Eclipse is IBM JDK.
- If you use Oracle JDK, ensure that the JDK configured in Eclipse is Oracle JDK.
- Do not use the same workspace and the sample project in the same path for different Eclipse programs.

----End

1.2.3 Downloading a Sample Project

Prerequisites

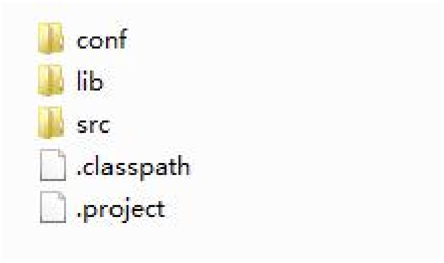
Ensure that CloudTable has been installed and is running properly.

Downloading a Sample Project

Step 1 Download the **Sample Code** project.

Step 2 After the download is complete, decompress the installation package of the sample code project to a local directory to obtain an Eclipse Java project. **Figure 1-2** shows the directory structure of the sample code project.

Figure 1-2 Directory structure of the sample code project



conf
lib
src
.classpath
.project

----End

Apache Maven Configuration

The sample project contains the HBase client JAR package. You can replace the JAR package with an open source HBase JAR package to access CloudTable. Open source HBase APIs later than 1.*X.X* are supported. If you need to import CloudTable's HBase JAR package to an application, configure the following dependencies in Maven:

```
<dependencies>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-client</artifactId>
        <version>1.3.1.0305-cloudtable</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-common</artifactId>
        <version>1.3.1.0305-cloudtable</version>
    </dependency>
</dependencies>
```

Use either of the following methods to configure the address of the mirror warehouse.

- **Configuration method 1**

  Add the address of the open source mirror warehouse to the mirrors in **setting.xml**.

```
<mirror>
    <id>repo2</id>
    <mirrorOf>central</mirrorOf>
    <url>https://repo1.maven.org/maven2/</url>
</mirror>
```

Add the following mirror warehouse address to the profiles in **setting.xml**.

```
<profile>
    <id>huaweicloudsdk</id>
    <repositories>
        <repository>
            <id>huaweicloudsdk</id>
            <url>https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
            <releases><enabled>true</enabled></releases>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
    </repositories>
</profile>
```

Add the following mirror warehouse address to the activeProfiles in **setting.xml**.

```
<activeProfile>huaweicloudsdk</activeProfile>
```

NOTE

The HUAWEI CLOUD open source mirror center does not provide third-party open source JAR files. After configuring HUAWEI CLOUD open source mirrors, you need to separately configure third-party Maven image repository address.

- **Configuration method 2**

  Add the following mirror warehouse address to the **pom.xml** file in the secondary development sample project.

```
    <repositories>
        <repository>
            <id>huaweicloudsdk</id>
            <url>https://mirrors.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
            <releases><enabled>true</enabled></releases>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>

        <repository>
            <id>central</id>
            <name>Mavn Centreal</name>
            <url>https://repo1.maven.org/maven2/</url>
        </repository>
    </repositories>
```

1.2.4 Configuring and Importing a Project

Background Information

After importing the CloudTable sample code project to Eclipse, you can start learning CloudTable application development samples.

Prerequisites

You have correctly configured the running environment. For details about how to configure the running environment, see **Preparing a Windows Running Environment**.

Procedure

Step 1 Import the sample project to the Windows development environment. For details about how to obtain the sample project, see **Downloading a Sample Project**.

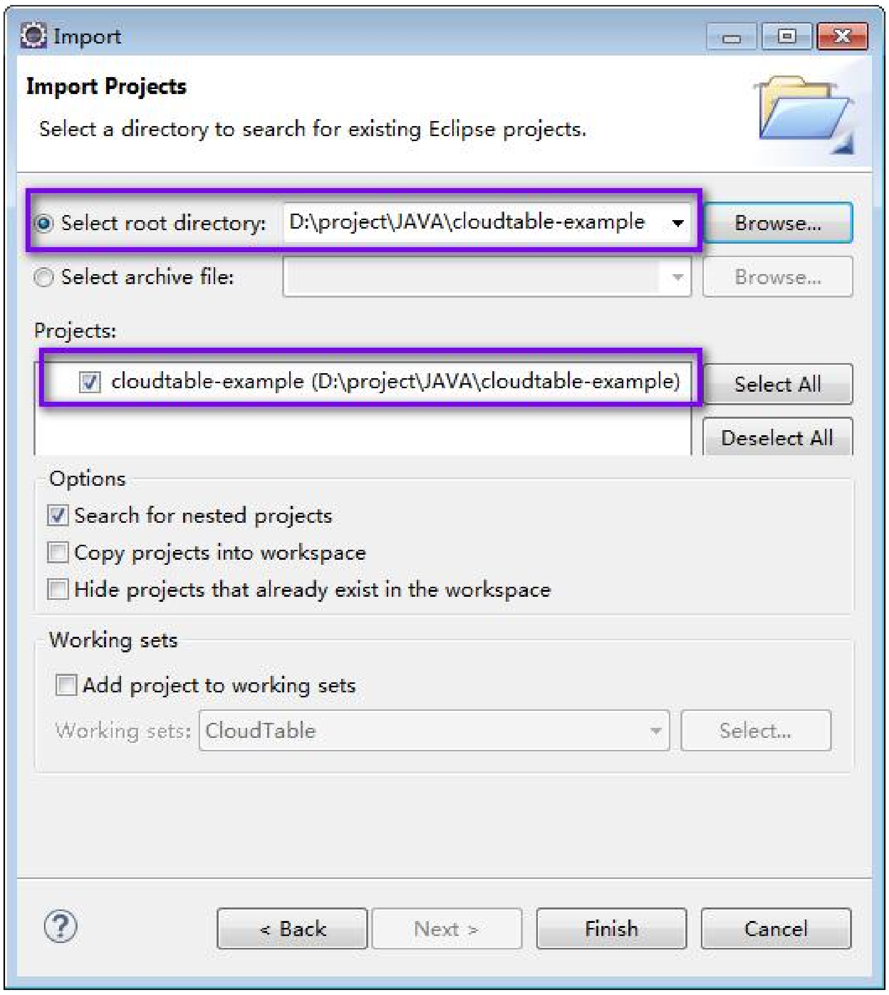
Step 2 In the application development environment, import the sample project to the Eclipse development environment.

1. Choose **File** > **Import** > **General** > **Existing Projects into Workspace** > **Next** > **Browse**.

   The **Browse Folder** dialog box is displayed, as shown in **Figure 1-3**.

2. Select the sample project folder, and click **Finish**.

Figure 1-3 Importing a sample project


Import

Import Projects

Select a directory to search for existing Eclipse projects.

Select root directory: D:\project\JAVA\cloudtable-example

Select archive file:

Projects:

☑ cloudtable-example (D:\project\JAVA\cloudtable-example)

Options

☑ Search for nested projects

☐ Copy projects into workspace

☐ Hide projects that already exist in the workspace

Working sets

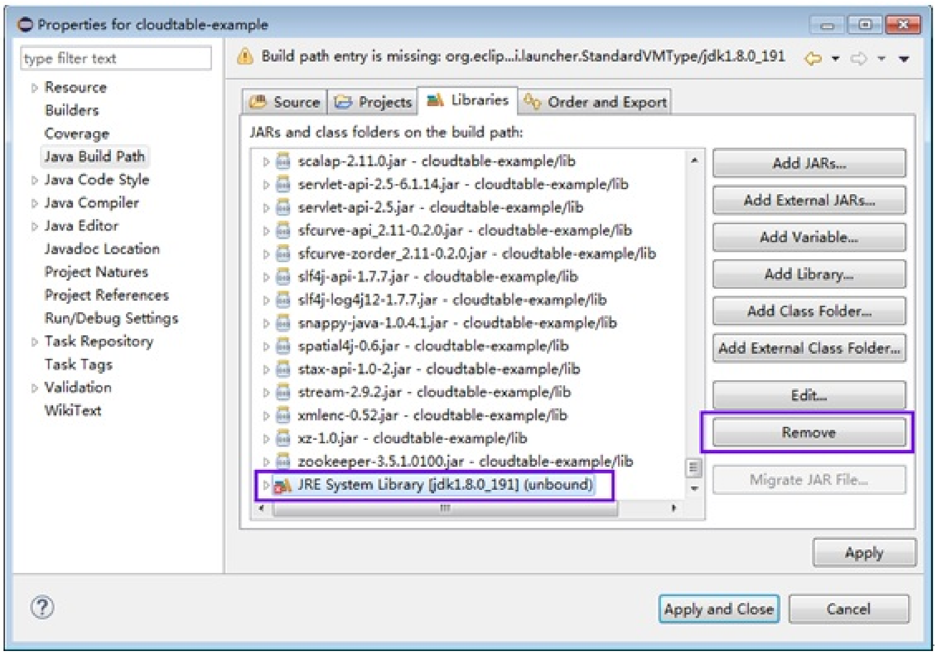
☐ Add project to working sets

Working sets: CloudTable

Step 3 Right-click the **cloudtable-example** project, and choose **Properties** from the shortcut menu. The **Properties for cloudtable-example** window is displayed.

1. In the navigation tree, select **Java Build Path**. Click the **Libraries** tab, select all error JDKs, and click **Remove**, as shown in **Figure 1-4**.

Figure 1-4 Delete error JDKs


Properties for cloudtable-example

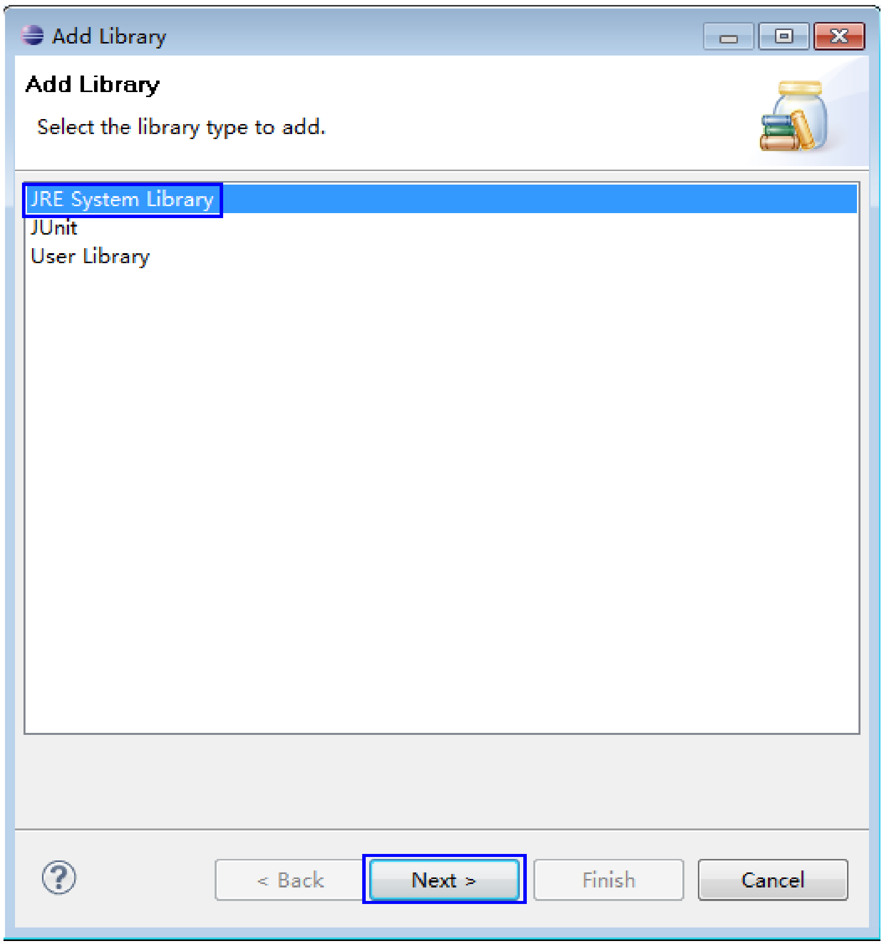
Build path entry is missing: org.eclip...i.launcher.StandardVMType/jdk1.8.0_191

JARs and class folders on the build path:

- scalap-2.11.0.jar - cloudtable-example/lib
- servlet-api-2.5-6.1.14.jar - cloudtable-example/lib
- servlet-api-2.5.jar - cloudtable-example/lib
- sfcurve-api_2.11-0.2.0.jar - cloudtable-example/lib
- sfcurve-zorder_2.11-0.2.0.jar - cloudtable-example/lib
- slf4j-api-1.7.7.jar - cloudtable-example/lib
- slf4j-log4j12-1.7.7.jar - cloudtable-example/lib
- snappy-java-1.0.4.1.jar - cloudtable-example/lib
- spatial4j-0.6.jar - cloudtable-example/lib
- stax-api-1.0-2.jar - cloudtable-example/lib
- stream-2.9.2.jar - cloudtable-example/lib
- xmlenc-0.52.jar - cloudtable-example/lib
- xz-1.0.jar - cloudtable-example/lib
- zookeeper-3.5.1.0100.jar - cloudtable-example/lib
- JRE System Library [jdk1.8.0_191] (unbound)

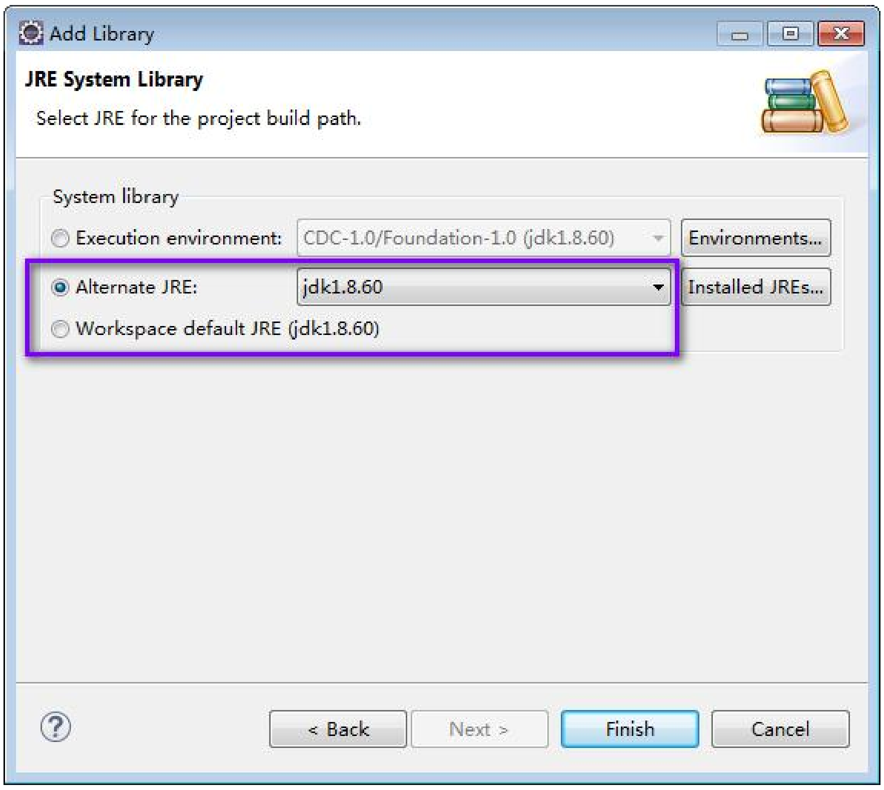
2. Click **Add Library...** shown in **Figure 1-5**. Select **JRE System Library** in the pop-up window.

Figure 1-5 Adding libraries



3. In the **Add Library** dialog box, select a JDK version from the drop-down list of **Alternate JRE** or **Workspace default JRE**. Select **Alternate JRE** and select the JDK version, as shown in **Figure 1-6**.

Figure 1-6 Selecting JRE



4. Click **Finish** to complete configuration and close the window.

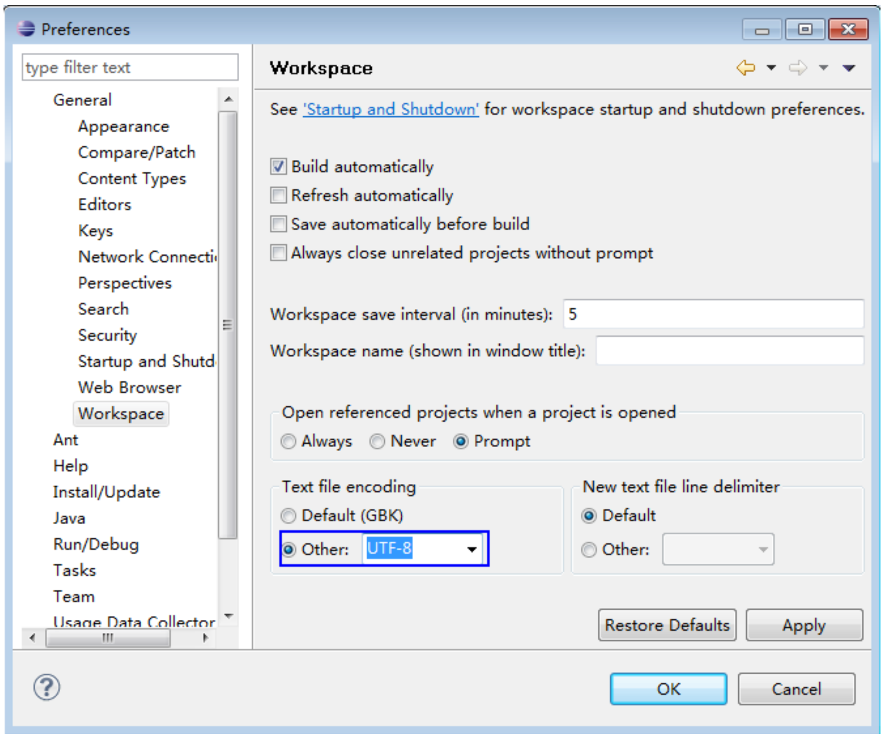
Step 4 Set the Eclipse text file coding format to prevent garbled characters.

1. On the Eclipse menu bar, choose **Window** > **Preferences**.

   The **Preferences** window is displayed.

2. In the navigation tree, choose **General** > **Workspace**. In the **Text file encoding** area, select **Other** and set the value to **UTF-8**. Click **Apply** and then **OK**. Figure 1-7 shows the settings.

Figure 1-7 Setting the Eclipse encoding format



See 'Startup and Shutdown' for workspace startup and shutdown preferences.

- [✓] Build automatically
- [ ] Refresh automatically
- [ ] Save automatically before build
- [ ] Always close unrelated projects without prompt

Workspace save interval (in minutes): 5
Workspace name (shown in window title):

Open referenced projects when a project is opened: Always Never [✓ Prompt]

Text file encoding: Default (GBK) [✓ Other: UTF-8]

New text file line delimiter: [✓ Default] Other:

Step 5 Open the **conf/hbase-site.xml** file in the sample project and change the value of **hbase.zookeeper.quorum** to the correct ZooKeeper address.

```
<property>
<name>hbase.zookeeper.quorum</name>
<value>xxx-zk1.cloudtable.com,xxx-zk2.cloudtable.com,xxx-zk3.cloudtable.com</value>
</property>
```

value is the domain name of the ZooKeeper cluster. Log in to the CloudTable console and choose **Cluster Management**. In the cluster list, locate the required cluster and obtain the address in the **Access Address (Intranet)** column.

----End

1.3 Developing HBase Applications

1.3.1 Typical Application Scenario

You can quickly learn and master the HBase development process and know key interface functions in a typical application scenario.

Description

Develop an application to manage information about users who use service A in an enterprise. **Table 1-3** provides the user information. Procedures are as follows:

- Create a user information table.
- Add users' educational backgrounds and titles to the table.
- Query user names and addresses by user ID.

- Query information by user name.
- Query information about users whose age ranges from 20 to 29.
- Collect the number of users and their maximum, minimum, and average age.
- Deregister users and delete user data from the user information table.
- Delete the user information table after service A ends.

Table 1-3 User information

ID	Name	Gender	Age	Address
12005000201	A	Male	19	Shenzhen, Guangdong
12005000202	B	Female	23	Shijiazhuang, Hebei
12005000203	C	Male	26	Ningbo, Zhejiang
12005000204	D	Male	18	Xiangyang, Hubei
12005000205	E	Female	21	Shangrao, Jiangxi
12005000206	F	Male	32	Zhuzhou, Hunan
12005000207	G	Female	29	Nanyang, Henan
12005000208	H	Female	30	Kaixian, Chongqing
12005000209	I	Male	26	Weinan, Shaanxi
12005000210	J	Male	25	Dalian, Liaoning

Data Planning

Proper design of a table structure, RowKeys, and column names enable you to make full use of HBase advantages. In the sample project, a unique ID is used as a RowKey, and columns are stored in the **info** column family.

1.3.2 Development Guideline

Function Description

Determine functions to be developed based on the preceding scenario. **Table 1-4** describes functions to be developed.

Table 1-4 Functions to be developed in HBase

No.	Procedure	Code Implementation
1	Create a table based on the information in **Typical Application Scenario**.	For details, see **Creating a Table**.
2	Import user data.	For details, see **Inserting Data**.
3	Add an educational background column family, and add educational backgrounds and titles to the user information table.	For details, see **Modifying a Table**.
4	Query user names and addresses by user ID.	For details, see **Reading Data Using Get**.
5	Query information by user name.	For details, see **Using a Filter**.
6	Deregister users and delete user data from the user information table.	For details, see **Deleting Data** .
7	Delete the user information table after service A ends.	For details, see **Deleting a Table**.

Key Design Principles

HBase is a distributed database system based on the lexicographic order of RowKeys. The RowKey design has great impact on performance, so the RowKeys must be designed based on specific services.

1.3.3 Sample Code Description

1.3.3.1 Parameter Configuration

Step 1 Before executing sample code, configure the correct ZooKeeper cluster address in the **hbase-site.xml** configuration file.

The configuration items are as follows:

```
<property>
<name>hbase.zookeeper.quorum</name>
<value>xxx-zk1.cloudtable.com,xxx-zk2.cloudtable.com,xxx-zk3.cloudtable.com</value>
</property>
```

value is the domain name of the ZooKeeper cluster. Log in to the CloudTable console and choose **Cluster Management**. In the cluster list, locate the required cluster and obtain the address in the **Access Address (Intranet)** column.

----End

1.3.3.2 Creating the Configuration Object

Function Description

HBase obtains configuration items by loading a configuration file.

NOTE

1. Loading the configuration file is time-consuming. If unnecessary, use the same Configuration object.
2. Multi-thread synchronization is not considered in the sample code. If necessary, add it by yourself. Other sample codes are the same.

Sample Code

The following code snippets are in the **com.huawei.cloudtable.hbase.examples** packet.

```
private static void init() throws IOException {
  // Default load from conf directory
  conf = HBaseConfiguration.create(); // Note [1]
  String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
  Path hbaseSite = new Path(userdir + "hbase-site.xml");
  if (new File(hbaseSite.toString()).exists()) {
    conf.addResource(hbaseSite);
  }
}
```

Precautions

- Note [1] If the **conf** directory of the configuration file is added to the **classpath** path, the code for loading the specified configuration file can be skipped.

1.3.3.3 Creating the Connection Object

Function Description

HBase creates a Connection object using the **ConnectionFactory.createConnection(configuration)** method. The transferred parameter is the Configuration created in the last step.

Connection encapsulates the connections between underlying applications and servers and ZooKeeper. Connection is instantiated using the **ConnectionFactory** class. Creating Connection is a heavyweight operation. Connection is thread-safe. Therefore, multiple client threads can share one Connection.

In a typical scenario, a client program uses a Connection, and each thread obtains its own Admin or Table instance and invokes the operation interface provided by the Admin or Table object. You are not advised to cache or pool Table and Admin. The lifecycle of Connection is maintained by invokers who can release resources by invoking **close()**.

NOTE

When the service code is connected to the same CloudTable cluster, you are advised to create one Connection and reuse it for multiple threads. You do not need to create a Connection for every thread. Connection is a connector for connecting to a CloudTable cluster. Excessive Connections will increase loads on ZooKeeper and deteriorate service read/write performance.

Sample Code

The following code snippet is an example of creating a Connection object:

```
private TableName tableName = null;
private Connection conn = null;

public HBaseSample(Configuration conf) throws IOException {
  this.tableName = TableName.valueOf("hbase_sample_table");
  this.conn = ConnectionFactory.createConnection(conf);
}
```

1.3.3.4 Creating a Table

Function Description

In HBase, a table is created using the **createTable** method of the **org.apache.hadoop.hbase.client.Admin** object. You need to specify a table name and a column family name. You can create a table by using either of the following methods, but the latter one is recommended:

- Quickly create a table. A newly created table contains only one region, which will be automatically split into multiple new regions as data increases.
- Create a table using pre-assigned regions. You can pre-assign multiple regions before creating a table. This mode accelerates data write at the beginning of massive data write.

NOTE

The table name and column family name of a table consist of letters, digits, and underscores (_) but cannot contain any special characters.

Sample Code

```
public void testCreateTable() {
  LOG.info("Entering testCreateTable.");

  // Specify the table descriptor.
  HTableDescriptor htd = new HTableDescriptor(tableName); // (1)

  // Set the column family name to info.
  HColumnDescriptor hcd = new HColumnDescriptor("info"); // (2)

  // Set data encoding methods. HBase provides DIFF,FAST_DIFF,PREFIX
  // and PREFIX_TREE
  hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);  // Note [1]

  // Set compression methods, HBase provides two default compression
  // methods:GZ and SNAPPY
  // GZ has the highest compression rate,but low compression and
  // decompression efficiency,fit for cold data
  // SNAPPY has low compression rate, but high compression and
  // decompression efficiency,fit for hot data.
```

```
 // it is advised to use SANPPY
 hcd.setCompressionType(Compression.Algorithm.SNAPPY);
 htd.addFamily(hcd); // (3)

 Admin admin = null;
 try {
   // Instantiate an Admin object.
   admin = conn.getAdmin(); // (4)
   if (!admin.tableExists(tableName)) {
     LOG.info("Creating table...");
     admin.createTable(htd); // Note [2] (5)
     LOG.info(admin.getClusterStatus());
     LOG.info(admin.listNamespaceDescriptors());
     LOG.info("Table created successfully.");
   } else {
     LOG.warn("table already exists");
   }
 } catch (IOException e) {
   LOG.error("Create table failed.", e);
 } finally {
   if (admin != null) {
     try {
       // Close the Admin object.
       admin.close();
     } catch (IOException e) {
       LOG.error("Failed to close admin ", e);
     }
   }
 }
 LOG.info("Exiting testCreateTable.");
}
```

Explanation

(1) Create a table descriptor.

(2) Create a column family descriptor.

(3) Add the column family descriptor to the table descriptor.

(4) Obtain the Admin object. You use the Admin object to create a table and a column family, check whether the table exists, modify the table structure and column family structure, and delete the table.

(5) Invoke the Admin object to create a table.

Precautions

- Note [1] Use the following code to set the compression mode for a column family:

```
// Set an encoding algorithm. HBase provides four encoding algorithms: DIFF, FAST_DIFF, PREFIX, and PREFIX_TREE.
 hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);

// Set a file compression mode. By default, HBase provides two compression algorithms: GZ and SNAPPY.
// GZ has a high compression rate but low compression and decompression performance. It is applicable to cold data.
 // SNAPPY has a low compression rate but high compression and decompression performance. It is applicable to hot data.
// It is recommended that SNAPPY compression be enabled by default.
 hcd.setCompressionType(Compression.Algorithm.SNAPPY);
```

- Note [2] Create a table by specifying the start and end RowKeys or pre-assigning regions using RowKey arrays. The code snippet is as follows:

```
// Create a table with pre-split regions.
 byte[][] splits = new byte[4][];
 splits[0] = Bytes.toBytes("A");
 splits[1] = Bytes.toBytes("H");
 splits[2] = Bytes.toBytes("O");
 splits[3] = Bytes.toBytes("U");
 admin.createTable(htd, splits);
```

1.3.3.5 Deleting a Table

Function Description

In HBase a table is deleted using the **deleteTable** method of **org.apache.hadoop.hbase.client.Admin**.

Sample Code

```
public void dropTable() {
  LOG.info("Entering dropTable.");
  Admin admin = null;
  try {
    admin = conn.getAdmin();
    if (admin.tableExists(tableName)) {
      // Disable the table before deleting it.
      admin.disableTable(tableName);
      // Delete table.
      admin.deleteTable(tableName);//Note [1]
    }
    LOG.info("Drop table successfully.");
  } catch (IOException e) {
    LOG.error("Drop table failed " ,e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Close admin failed " ,e);
      }
    }
  }
  LOG.info("Exiting dropTable.");
}
```

Precautions

Note [1] Only after the **disableTable** API is called, the table can be deleted by calling the **deleteTable** API. Therefore, **deleteTable** is often used together with **disableTable**, **enableTable**, **tableExists**, **isTableEnabled**, and **isTableDisabled**.

1.3.3.6 Modifying a Table

Function Description

In HBase, table information is modified using the **modifyTable** method of **org.apache.hadoop.hbase.client.Admin**.

Sample Code

```
public void testModifyTable() {
  LOG.info("Entering testModifyTable.");
  // Specify the column family name.
  byte[] familyName = Bytes.toBytes("education");
  Admin admin = null;
  try {
    // Instantiate an Admin object.
    admin = conn.getAdmin();
    // Obtain the table descriptor.
    HTableDescriptor htd = admin.getTableDescriptor(tableName);
    // Check whether the column family is specified before modification.
    if (!htd.hasFamily(familyName)) {
      // Create the column descriptor.
      HColumnDescriptor hcd = new HColumnDescriptor(familyName);
      htd.addFamily(hcd);
      // Disable the table to get the table offline before modifying
      // the table.
      admin.disableTable(tableName);
      // Submit a modifyTable request.
      admin.modifyTable(tableName, htd);  //Note [1]
      // Enable the table to get the table online after modifying the
      // table.
      admin.enableTable(tableName);
    }
    LOG.info("Modify table successfully.");
  } catch (IOException e) {
    LOG.error("Modify table failed " ,e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Close admin failed " ,e);
      }
    }
  }
  LOG.info("Exiting testModifyTable.");
}
```

Precautions

Note [1] Only after the **disableTable** API is called, the table can be modified by calling the **modifyTable** API. Then, call the **enableTable** API to enable the table again.

1.3.3.7 Inserting Data

Function Description

HBase is a column-based database. A row of data may have multiple column families, and a column family may contain multiple columns. When writing data, you must specify the columns (including the column family names and column names) to which data is written. In HBase, data (a row of data or data sets) is inserted using the **put** method of HTable.

Sample Code

```
public void testPut() {
  LOG.info("Entering testPut.");
  // Specify the column family name.
  byte[] familyName = Bytes.toBytes("info");
  // Specify the column name.
  byte[][] qualifiers = { Bytes.toBytes("name"), Bytes.toBytes("gender"),
      Bytes.toBytes("age"), Bytes.toBytes("address") };
```

```
Table table = null;
try {
  // Instantiate an HTable object.
  table = conn.getTable(tableName);
  List<Put> puts = new ArrayList<Put>();
  // Instantiate a Put object.
  Put put = new Put(Bytes.toBytes("012005000201"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("A"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("19"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shenzhen, Guangdong"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000202"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("B"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("23"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shijiazhuang, Hebei"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000203"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("C"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("26"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Ningbo, Zhejiang"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000204"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("D"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("18"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Xiangyang, Hubei"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000205"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("E"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("21"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shangrao, Jiangxi"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000206"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("F"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("32"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Zhuzhou, Hunan"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000207"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("G"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("29"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Nanyang, Henan"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000208"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("H"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("30"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Kaixian, Chongqing"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000209"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("I"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("26"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Weinan, Shaanxi"));
  puts.add(put);
  put = new Put(Bytes.toBytes("012005000210"));
  put.addColumn(familyName, qualifiers[0], Bytes.toBytes("J"));
  put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
  put.addColumn(familyName, qualifiers[2], Bytes.toBytes("25"));
  put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Dalian, Liaoning"));
  puts.add(put);
  // Submit a put request.
  table.put(puts);
```

```
    LOG.info("Put successfully.");
  } catch (IOException e) {
    LOG.error("Put failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testPut.");
}
```

Precautions

Multiple threads are not allowed to use the same HTable instance at the same time. HTable is a non-thread-safe class. If an HTable instance is used by multiple threads at the same time, exceptions will occur.

1.3.3.8 Deleting Data

Function Description

In HBase, data (a row of data or data sets) is deleted using the **delete** method of a Table instance.

Sample Code

```
public void testDelete() {
  LOG.info("Entering testDelete.");

  byte[] rowKey = Bytes.toBytes("012005000201");

  Table table = null;
  try {
    // Instantiate an HTable object.
    table = conn.getTable(tableName);

    // Instantiate a Delete object.
    Delete delete = new Delete(rowKey);

    // Submit a delete request.
    table.delete(delete);

    LOG.info("Delete table successfully.");
  } catch (IOException e) {
    LOG.error("Delete table failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testDelete.");
}
```

1.3.3.9 Reading Data Using Get

Function Description

Before reading data from a table, create a table instance and a Get object. You can also set parameters for the Get object, such as the column family name and column name. Query results are stored in the Result object that stores multiple Cells.

Sample Code

```
public void testGet() {
  LOG.info("Entering testGet.");
  // Specify the column family name.
  byte[] familyName = Bytes.toBytes("info");
  // Specify the column name.
  byte[][] qualifier = { Bytes.toBytes("name"), Bytes.toBytes("address") };
  // Specify RowKey.
  byte[] rowKey = Bytes.toBytes("012005000201");
  Table table = null;
  try {
    // Create the Table instance.
    table = conn.getTable(tableName);
    // Instantiate a Get object.
    Get get = new Get(rowKey);
    // Set the column family name and column name.
    get.addColumn(familyName, qualifier[0]);
    get.addColumn(familyName, qualifier[1]);
    // Submit a get request.
    Result result = table.get(get);
    // Print query results.
    for (Cell cell : result.rawCells()) {
      LOG.info(Bytes.toString(CellUtil.cloneRow(cell)) + ":"
          + Bytes.toString(CellUtil.cloneFamily(cell)) + ","
          + Bytes.toString(CellUtil.cloneQualifier(cell)) + ","
          + Bytes.toString(CellUtil.cloneValue(cell)));
    }
    LOG.info("Get data successfully.");
  } catch (IOException e) {
    LOG.error("Get data failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testGet.");
}
```

1.3.3.10 Reading Data Using Scan

Function Description

Before reading data from a table, instantiate the Table instance of the table, and then create a Scan object and set parameters for the Scan object based on search criteria. To improve query efficiency, you are advised to specify StartRow and StopRow. Query results are stored in the ResultScanner object, where each row of data is stored as a Result object that stores multiple Cells.

Sample Code

```
public void testScanData() {
  LOG.info("Entering testScanData.");
  Table table = null;
  // Instantiate a ResultScanner object.
  ResultScanner rScanner = null;
  try {
    // Create the Configuration instance.
    table = conn.getTable(tableName);
    // Instantiate a Get object.
    Scan scan = new Scan();
    scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
    // Set the cache size.
    scan.setCaching(1000);
    // Submit a scan request.
    rScanner = table.getScanner(scan);
    // Print query results.
    for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
      for (Cell cell : r.rawCells()) {
        LOG.info(Bytes.toString(CellUtil.cloneRow(cell)) + ":"
            + Bytes.toString(CellUtil.cloneFamily(cell)) + ","
            + Bytes.toString(CellUtil.cloneQualifier(cell)) + ","
            + Bytes.toString(CellUtil.cloneValue(cell)));
      }
    }
    LOG.info("Scan data successfully.");
  } catch (IOException e) {
    LOG.error("Scan data failed " ,e);
  } finally {
    if (rScanner != null) {
      // Close the scanner object.
      rScanner.close();
    }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testScanData.");
}
```

Precautions

1. You are advised to specify StartRow and StopRow to ensure good performance with a specified Scan scope.
2. You can set **Batch** and **Caching**.
   - Batch

     Batch indicates the maximum number of records returned each time when the **next** API is invoked using Scan. This parameter is related to the number of columns read each time.
   - Caching

     Caching indicates the maximum number of next records returned for a remote procedure call (RPC) request. This parameter is related to the number of rows read by an RPC.

1.3.3.11 Using a Filter

Function Description

HBase Filter is used to filter data during Scan and Get. You can specify the filter criteria, such as filtering by RowKey, column name, or column value.

Sample Code

```
public void testSingleColumnValueFilter() {
  LOG.info("Entering testSingleColumnValueFilter.");
  Table table = null;
  ResultScanner rScanner = null;

  try {
    table = conn.getTable(tableName);
    Scan scan = new Scan();
    scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
    // Set the filter criteria.
    SingleColumnValueFilter filter = new SingleColumnValueFilter(
        Bytes.toBytes("info"), Bytes.toBytes("name"), CompareOp.EQUAL,
        Bytes.toBytes("I"));
    scan.setFilter(filter);
    // Submit a scan request.
    rScanner = table.getScanner(scan);
    // Print query results.
    for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
      for (Cell cell : r.rawCells()) {
        LOG.info(Bytes.toString(CellUtil.cloneRow(cell)) + ":"
            + Bytes.toString(CellUtil.cloneFamily(cell)) + ","
            + Bytes.toString(CellUtil.cloneQualifier(cell)) + ","
            + Bytes.toString(CellUtil.cloneValue(cell)));
      }
    }
    LOG.info("Single column value filter successfully.");
  } catch (IOException e) {
    LOG.error("Single column value filter failed " ,e);
  } finally {
      if (rScanner != null) {
          // Close the scanner object.
          rScanner.close();
        }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testSingleColumnValueFilter.");
}
```

1.4 Accessing HBase ThriftServer

1.4.1 Accessing the ThriftServer Operation Table

Scenario

After importing the host where the ThriftServer instances are located and the port that provides services, you can create a Thrift client using the authentication credential and configuration file, access ThriftServer, and obtain table names, create a table, and delete a table based on the specified namespace.

Procedure

Step 1 Log in to the CloudTable console.

Step 2 Select a region in the upper left corner of the page.

Step 3 Click **Cluster Management** to go to the cluster management page.

Step 4 Click the name of an HBase cluster to go to the cluster details page and check the Thrift Server status. If Thrift Server is enabled, no further action is required. If Thrift Server is disabled, return to the cluster management page and choose **More** > **Enable Thrift Server**.

NOTE

- Currently, Thrift Server cannot interconnect with ELB. A single client does not support multiple threads.
- Download the Thrift installation package from https:*//www.apache.org/dyn/closer.cgi?path=/thrift/0.11.0/thrift-0.11.0.tar.gz*.
- Download the ThriftServer code example package from *https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-3.3.0/src/hbase-examples/hbase-thrift-example*, decompress the package, and obtain the code example.

----End

Java Code Example

- Preparing for compilation

  Add the following configuration to the **pom.xml** file of the **hbase-thrift-example** project:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>2.3</version>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.huawei.bigdata.hbase.examples.TestMain</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
            <executions>
                <execution>
                    <id>make-assemble</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

- Initializing configuration

  The following code snippets belong to the **TestMain** class in the **com.huawei.bigdata.hbase.examples** package of the **hbase-thrift-example** sample project.

```
    private static void init() throws IOException {
        // Default load from conf directory
        conf = HBaseConfiguration.create();

        String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
[1]
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
        conf.addResource(new Path(userdir + "core-site.xml"), false);
        conf.addResource(new Path(userdir + "hdfs-site.xml"), false);
        conf.addResource(new Path(userdir + "hbase-site.xml"), false);
}
```

  [1] **userdir** obtains the **conf** directory in the resource path after compilation. The **core-site.xml**, **hdfs-site.xml**, and **hbase-site.xml** files used for initial configuration must be stored in the **src/main/resources/conf** directory.

- Connecting to a ThriftServer instance

  The following code snippets belong to the **TestMain** class in the **com.huawei.bigdata.hbase.examples** package of the **hbase-thrift-example** sample project.

```
    try {
        test = new ThriftSample();
        test.test("xxx.xxx.xxx.xxx", THRIFT_PORT, conf);[2]
    } catch (TException | IOException e) {
        LOG.error("Test thrift error", e);
    }
```

  [2] The value of the input parameter **test.test()** is the IP address of the node where the ThriftServer instance to be accessed is located. Change the IP address to the actual one. The IP address of the node must be configured in the hosts file of the local host where the sample code is run.

  THRIFT_PORT is the value of **hbase.regionserver.thrift.port** configured for the ThriftServer instance.

- Invoking methods

```
// Get table of specified namespace.
getTableNamesByNamespace(client, "default");
// Create table.
createTable(client, TABLE_NAME);
// Delete specified table.
 deleteTable(client, TABLE_NAME);
```

- Obtaining table names based on the specified namespace

  The following code snippets are in the **getTableNamesByNamespace** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void getTableNamesByNamespace(THBaseService.Iface client, String namespace) throws
TException {
     client.getTableNamesByNamespace(namespace)
         .forEach(
             tTableName -> LOGGER.info("{}", TableName.valueOf(tTableName.getNs(),
tTableName.getQualifier())));
 }
```

- Creating a table

  The following code snippets are in the **createTable** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void createTable(THBaseService.Iface client, String tableName) throws TException,
IOException {
    TTableName table = getTableName(tableName);
    TTableDescriptor descriptor = new TTableDescriptor(table);
    descriptor.setColumns(
        Collections.singletonList(new
TColumnFamilyDescriptor().setName(COLUMN_FAMILY.getBytes())));
    if (client.tableExists(table)) {
        LOGGER.warn("Table {} is exists, delete it.", tableName);
        client.disableTable(table);
        client.deleteTable(table);
    }
    client.createTable(descriptor, null);
    if (client.tableExists(table)) {
        LOGGER.info("Created {}.", tableName);
    } else {
        LOGGER.error("Create {} failed.", tableName);
    }
 }
```

- Deleting a table

  The following code snippets are in the **deleteTable** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void deleteTable(THBaseService.Iface client, String tableName) throws TException,
IOException {
    TTableName table = getTableName(tableName);
    if (client.tableExists(table)) {
        client.disableTable(table);
        client.deleteTable(table);
        LOGGER.info("Deleted {}.", tableName);
    } else {
        LOGGER.warn("{} not exist.", tableName);
    }
 }
```

C++ Code Example

- Install the Thrift application on the client node and log in to the SSH tool debugging environment.

```
[192.168.0.82_ testThrift]#cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
[192.168.0.82_ testThrift]#thrift -version
Thrift version 0.11.0
```

- In the **/opt** directory on the client node, create the **hbase.thrift** file. The file content is as follows:

```
namespace java org.apache.hadoop.hbase.thrift2.generated
namespace cpp apache.hadoop.hbase.thrift2
namespace rb Apache.Hadoop.Hbase.Thrift2
namespace py hbase
namespace perl Hbase

struct TTimeRange {
  1: required i64 minStamp,
  2: required i64 maxStamp
}
struct TColumn {
  1: required binary family,
  2: optional binary qualifier,
  3: optional i64 timestamp
}
struct TColumnValue {
  1: required binary family,
  2: required binary qualifier,
  3: required binary value,
  4: optional i64 timestamp,
  5: optional binary tags
}
struct TColumnIncrement {
  1: required binary family,
  2: required binary qualifier,
  3: optional i64 amount = 1
}
struct TResult {
  1: optional binary row,
  2: required list<TColumnValue> columnValues
}
enum TDeleteType {
  DELETE_COLUMN = 0,
  DELETE_COLUMNS = 1
}
enum TDurability {
  SKIP_WAL = 1,
  ASYNC_WAL = 2,
  SYNC_WAL = 3,
  FSYNC_WAL = 4
}
struct TAuthorization {
 1: optional list<string> labels
}
struct TCellVisibility {
 1: optional string expression
}
struct TGet {
  1: required binary row,
  2: optional list<TColumn> columns,

  3: optional i64 timestamp,
  4: optional TTimeRange timeRange,

  5: optional i32 maxVersions,
  6: optional binary filterString,
  7: optional map<binary, binary> attributes
  8: optional TAuthorization authorizations
}
struct TPut {
  1: required binary row,
  2: required list<TColumnValue> columnValues
  3: optional i64 timestamp,
  5: optional map<binary, binary> attributes,
  6: optional TDurability durability,
  7: optional TCellVisibility cellVisibility
}
struct TDelete {
  1: required binary row,
  2: optional list<TColumn> columns,
  3: optional i64 timestamp,
  4: optional TDeleteType deleteType = 1,
  6: optional map<binary, binary> attributes,
  7: optional TDurability durability
}
struct TIncrement {
  1: required binary row,
  2: required list<TColumnIncrement> columns,
  4: optional map<binary, binary> attributes,
  5: optional TDurability durability
  6: optional TCellVisibility cellVisibility
}
struct TAppend {
  1: required binary row,
  2: required list<TColumnValue> columns,
  3: optional map<binary, binary> attributes,
  4: optional TDurability durability
  5: optional TCellVisibility cellVisibility
```

```
}
struct TScan {
  1: optional binary startRow,
  2: optional binary stopRow,
  3: optional list<TColumn> columns
  4: optional i32 caching,
  5: optional i32 maxVersions=1,
  6: optional TTimeRange timeRange,
  7: optional binary filterString,
  8: optional i32 batchSize,
  9: optional map<binary, binary> attributes
  10: optional TAuthorization authorizations
  11: optional bool reversed
  12: optional bool cacheBlocks
}
union TMutation {
  1: TPut put,
  2: TDelete deleteSingle,
}
struct TRowMutations {
  1: required binary row
  2: required list<TMutation> mutations
}
struct THRegionInfo {
  1: required i64 regionId
  2: required binary tableName
  3: optional binary startKey
  4: optional binary endKey
  5: optional bool offline
  6: optional bool split
  7: optional i32 replicaId
}
struct TServerName {
  1: required string hostName
  2: optional i32 port
  3: optional i64 startCode
}
struct THRegionLocation {
  1: required TServerName serverName
  2: required THRegionInfo regionInfo
}
enum TCompareOp {
  LESS = 0,
  LESS_OR_EQUAL = 1,
  EQUAL = 2,
  NOT_EQUAL = 3,
  GREATER_OR_EQUAL = 4,
  GREATER = 5,
  NO_OP = 6
}
exception TIOError {
  1: optional string message
}
exception TIllegalArgument {
  1: optional string message
}
service THBaseService {
  bool exists(
    1: required binary table,
    2: required TGet tget
  ) throws (1:TIOError io)
  TResult get(
    1: required binary table,
    2: required TGet tget
  ) throws (1: TIOError io)
  list<TResult> getMultiple(
    1: required binary table,
    2: required list<TGet> tgets
  ) throws (1: TIOError io)
```

```
void put(
  1: required binary table,
  2: required TPut tput
) throws (1: TIOError io)
bool checkAndPut(
  1: required binary table,
  2: required binary row,
  3: required binary family,
  4: required binary qualifier,
  5: binary value,
  6: required TPut tput
) throws (1: TIOError io)
void putMultiple(
  1: required binary table,
  2: required list<TPut> tputs
) throws (1: TIOError io)
void deleteSingle(
  1: required binary table,
  2: required TDelete tdelete
) throws (1: TIOError io)
list<TDelete> deleteMultiple(
  1: required binary table,
  2: required list<TDelete> tdeletes
) throws (1: TIOError io)
bool checkAndDelete(
  1: required binary table,
  2: required binary row,
  3: required binary family,
  4: required binary qualifier,
  5: binary value,
  6: required TDelete tdelete
) throws (1: TIOError io)
TResult increment(
  1: required binary table,
  2: required TIncrement tincrement
) throws (1: TIOError io)
TResult append(
  1: required binary table,
  2: required TAppend tappend
) throws (1: TIOError io)
i32 openScanner(
  1: required binary table,
  2: required TScan tscan,
) throws (1: TIOError io)
list<TResult> getScannerRows(
  1: required i32 scannerId,
  2: i32 numRows = 1
) throws (
  1: TIOError io,
  2: TIllegalArgument ia
)
void closeScanner(
  1: required i32 scannerId
) throws (
  1: TIOError io,
  2: TIllegalArgument ia
)
void mutateRow(
  1: required binary table,
  2: required TRowMutations trowMutations
) throws (1: TIOError io)
list<TResult> getScannerResults(
  1: required binary table,
  2: required TScan tscan,
  3: i32 numRows = 1
) throws (
  1: TIOError io
)
THRegionLocation getRegionLocation(
    1: required binary table,
    2: required binary row,
    3: bool reload,
  ) throws (
    1: TIOError io
  )
  list<THRegionLocation> getAllRegionLocations(
    1: required binary table,
  ) throws (
    1: TIOError io
  )
  bool checkAndMutate(
    1: required binary table,
    2: required binary row,
    3: required binary family,
    4: required binary qualifier,
    5: required TCompareOp compareOp,
    6: binary value,
    7: required TRowMutations rowMutations
  ) throws (1: TIOError io)
}
```

- Run the **thrift --gen cpp /opt/hbase.thrift** command. After the command is successfully executed, the **gen-cpp** directory is generated in the **/opt** directory.
- In the **/opt** directory on the client node, create the **Makefile** file. The file content is as follows:

```
THRIFT_DIR = /usr/local/include/thrift
LIB_DIR = /usr/local/lib

GEN_SRC =  ./gen-cpp/hbase_types.cpp ./gen-cpp/hbase_constants.cpp ./gen-cpp/THBaseService.cpp

.PHONY: clean help

default: HbaseClient
HbaseClient: HbaseClient.cpp
    g++  -o HbaseClient -I${THRIFT_DIR}  -I./gen-cpp -L${LIB_DIR} -Wl,-rpath=${LIB_DIR} HbaseClient.cpp $
{GEN_SRC} -lthrift -g

clean:
    rm -rf HbaseClient

help:
    $(warning "Makefile for C++ Hbase Thrift HbaseClient. Modify THRIFT_DIR and LIB_DIR in the \
file to point to correct locations. See $${HBASE_ROOT}/hbase-examples/README.txt for \
details.")
    @:
```

- In the **/opt** directory on the client node, create the **HbaseClient.cpp** file. The file content is as follows:

```
#include "THBaseService.h"
#include <config.h>
#include <vector>
#include <ostream>
#include <iostream>
#include "transport/TSocket.h"
#include <transport/TBufferTransports.h>
#include <protocol/TBinaryProtocol.h>

using namespace std;
using namespace apache::thrift;
using namespace apache::thrift::protocol;
using namespace apache::thrift::transport;
using namespace apache::hadoop::hbase::thrift2;

using boost::shared_ptr;

int readdb(int argc, char** argv) {
   fprintf(stderr, "readdb start\n");
```

```
        int port = atoi(argv[2]);
        boost::shared_ptr<TSocket> socket(new TSocket(argv[1], port));
        boost::shared_ptr<TTransport> transport(new TBufferedTransport(socket));
        boost::shared_ptr<TProtocol> protocol(new TBinaryProtocol(transport));
        try {
        transport->open();
        printf("open\n");
        THBaseServiceClient  client(protocol);
        TResult tresult;
        TGet get;

        std::vector<TColumnValue> cvs;
        const std::string table("test");
        const std::string thisrow="row-1-1";
        get.__set_row(thisrow);
        bool be = client.exists(table,get);
        printf("exists result value = %d\n", be);
        client.get(tresult,table,get);
        vector<TColumnValue> list=tresult.columnValues;
        std::vector<TColumnValue>::const_iterator iter;
        for(iter=list.begin();iter!=list.end();iter++) {
            printf("%s, %s,%s\n",(*iter).family.c_str(),(*iter).qualifier.c_str(),(*iter).value.c_str());
        }
        transport->close();
        printf("close\n");
     } catch (const TException &tx) {
        std::cerr << "ERROR(exception): " << tx.what() << std::endl;
     }
    fprintf(stderr, "readdb stop\n");
    return 0;
    return 0;
}

int writedb(int argc, char** argv){
   fprintf(stderr, "writedb start\n");
        int port = atoi(argv[2]);
        boost::shared_ptr<TSocket> socket(new TSocket(argv[1], port));
        boost::shared_ptr<TTransport> transport(new TBufferedTransport(socket));
        boost::shared_ptr<TProtocol> protocol(new TBinaryProtocol(transport));
        try {
        char buf[128];
        transport->open();
        printf("open\n");
        THBaseServiceClient  client(protocol);
        TResult tresult;
        TGet get;
        std::vector<TPut> puts;
        const std::string table("test");
        for(int i = 0; i < 10; i++) {
            fprintf(stderr, "%d, ", i);
            for(int j = 0; j < 10; j++) {
                TPut put;
                std::vector<TColumnValue> cvs;
                //put data
                sprintf(buf, "row-%d-%d", i, j);
                const std::string thisrow(buf);
                put.__set_row(thisrow);
                TColumnValue tcv;
                tcv.__set_family("info");
                tcv.__set_qualifier("age");
                sprintf(buf, "%d", i * j);
                tcv.__set_value(buf);
                cvs.insert(cvs.end(), tcv);
                put.__set_columnValues(cvs);
                puts.insert(puts.end(), put);
            }
            client.putMultiple(table, puts);
            puts.clear();
        }

        transport->close();
        printf("close\n");
     } catch (const TException &tx) {
        std::cerr << "ERROR(exception): " << tx.what() << std::endl;
     }
    fprintf(stderr, "writedb stop\n");
    return 0;
}

int main(int argc, char **argv) {
    if(argc != 3) {
        fprintf(stderr, "param  is :XX ip port\n");
        return -1;
    }
    writedb(argc, argv);
    readdb(argc, argv);
    return 0;
}
```

- Then, run the **make** command in the **/opt** directory to compile the provided C ++ example code.
- Go to the HBase shell and run the **create'test', 'info'** command to create a test table.
- Run the **./HbaseClient** *thrift2_server_ip thrift_server_port* command, for example, **./HbaseClient 192.168.5.238 9090**, to check whether the test table can be read and written. The verification result is as follows:

```
[192.168.0.82_ thrift2]#./HbaseClient 192.168.5.238 9090
writedb start
open
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, close
writedb stop
readdb start
open
exists result value = 1
info, age,1
close
readdb stop
[192.168.0.82_ thrift2]#
```

1.4.2 Accessing Thrift Server to Read Data

Function Description

Input the host where the ThriftServer instance is located and the port that provides services, create a Thrift client, access the Thrift Server instance, and use the **GET** and **Scan** commands to read data.

Example Code

- Invoking methods

```
// Get data with single get.
getData(client, TABLE_NAME);

// Get data with getlist.
getDataList(client, TABLE_NAME);

// Scan data.
 scanData(client, TABLE_NAME);
```

- Using the **get** method to write data.

The following code snippets are in the **getData** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void getData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test getData.");
    TGet get = new TGet();
    get.setRow("row1".getBytes());

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    TResult result = client.get(table, get);
    printResult(result);
    LOGGER.info("Test getData done.");
}
```

- Using getlist to read data

  The following code snippets are in the **getDataList** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void getDataList(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test getDataList.");
    List<TGet> getList = new ArrayList<>();
    TGet get1 = new TGet();
    get1.setRow("row1".getBytes());
    getList.add(get1);

    TGet get2 = new TGet();
    get2.setRow("row2".getBytes());
    getList.add(get2);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    List<TResult> resultList = client.getMultiple(table, getList);
    for (TResult tResult : resultList) {
        printResult(tResult);
    }
    LOGGER.info("Test getDataList done.");
}
```

- Using the **scan** method to write data

  The following code snippets are in the **scanData** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void scanData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test scanData.");
    int scannerId = -1;
    try {
        ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
        TScan scan = new TScan();
        scan.setLimit(500);
        scannerId = client.openScanner(table, scan);
        List<TResult> resultList = client.getScannerRows(scannerId, 100);
        while (resultList != null && !resultList.isEmpty()) {
            for (TResult tResult : resultList) {
                printResult(tResult);
            }
            resultList = client.getScannerRows(scannerId, 100);
        }
    } finally {
        if (scannerId != -1) {
            client.closeScanner(scannerId);
            LOGGER.info("Closed scanner {}.", scannerId);
        }
    }
    LOGGER.info("Test scanData done.");
}
```

1.4.3 Accessing Thrift Server to Write Data

Function Description

Input the host where the ThriftServer instance is located and the port that provides services, create a Thrift client, access the Thrift Server instance, and use the **put** and **putMultiple** commands to write data.

Example Code

- Invoking methods

```
// Write data with put.
putData(client, TABLE_NAME);

// Write data with putlist.
 putDataList(client, TABLE_NAME);
```

- Using **Put** to write data

  The following code snippets are in the **putData** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void putData(THBaseService.Iface client, String tableName) throws TException {
     LOGGER.info("Test putData.");
     TPut put = new TPut();
     put.setRow("row1".getBytes());

     TColumnValue columnValue = new TColumnValue();
     columnValue.setFamily(COLUMN_FAMILY.getBytes());
     columnValue.setQualifier("q1".getBytes());
     columnValue.setValue("test value".getBytes());
     List<TColumnValue> columnValues = new ArrayList<>(1);
     columnValues.add(columnValue);
     put.setColumnValues(columnValues);

     ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
     client.put(table, put);
     LOGGER.info("Test putData done.");
 }
```

- Using **putMultiple** to write data

  The following code snippets are in the **putDataList** method in the **ThriftSample** class of the **hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples** packet.

```
private void putDataList(THBaseService.Iface client, String tableName) throws TException {
     LOGGER.info("Test putDataList.");
     TPut put1 = new TPut();
     put1.setRow("row2".getBytes());
     List<TPut> putList = new ArrayList<>();

     TColumnValue q1Value = new TColumnValue(ByteBuffer.wrap(COLUMN_FAMILY.getBytes()),
         ByteBuffer.wrap("q1".getBytes()), ByteBuffer.wrap("test value".getBytes()));
     TColumnValue q2Value = new TColumnValue(ByteBuffer.wrap(COLUMN_FAMILY.getBytes()),
         ByteBuffer.wrap("q2".getBytes()), ByteBuffer.wrap("test q2 value".getBytes()));
     List<TColumnValue> columnValues = new ArrayList<>(2);
     columnValues.add(q1Value);
     columnValues.add(q2Value);
     put1.setColumnValues(columnValues);
     putList.add(put1);

     TPut put2 = new TPut();
     put2.setRow("row3".getBytes());

     TColumnValue columnValue = new TColumnValue();
```

```
    columnValue.setFamily(COLUMN_FAMILY.getBytes());
    columnValue.setQualifier("q1".getBytes());
    columnValue.setValue("test q1 value".getBytes());
    put2.setColumnValues(Collections.singletonList(columnValue));
    putList.add(put2);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    client.putMultiple(table, putList);
    LOGGER.info("Test putDataList done.");
}
```

1.5 Developing Term Index Applications

1.5.1 Application Background

CloudTable is a big data storage service that provides efficient key-value random query. On this basis, CloudTable introduces self-developed distributed multidimensional term index feature. The storage format and computing are based on a bitmap. You can define which fields in HBase need to build a term index based on service requirements. Term data is automatically generated when you write data. In addition, the term index provides efficient multidimensional term query APIs based on the Lucene syntax. The APIs are applicable to scenarios such as user profile, recommendation system, AI, and spatiotemporal data analysis.

CloudTable supports a term index. You only need to create a CloudTable cluster to develop a client application on an ECS for a multidimensional term query.

1.5.2 Typical Application Scenario

You can quickly learn and master the term index development process and know key API functions in a typical application scenario.

Description

An online learning app (not for free) adds various attribute terms to members to facilitate subsequent resource placement and precise marketing. For example, the app needs to count users who have a bachelor's degree and a master's degree in milliseconds and obtain specific user information.

The fields in the user information table are as follows:

Table 1-5 User information

Field	Field Description	Term Index Required
name	User name	No
education	User's education background	Yes
otherInfo	Other user information	No

1.5.3 Development Guideline

Table 1-6 Development guideline

No.	Procedure	Code Implementation
1	Enable a term index when creating an HBase table.	For details, see **Enabling a Term Index When Creating a Table**.
2	Write data using HBase Put.	For details, see **Writing Data**.
3	Query data.	For details, see the following sections: ● **Common Query** ● **Sampling Query** ● **Paging Query** ● **Statistics Query**

1.5.4 Sample Code Description

1.5.4.1 Parameter Configuration

Step 1 Before executing sample code, configure the correct ZooKeeper cluster address in the **hbase-site.xml** configuration file.

The configuration items are as follows:

```
<property>
<name>hbase.zookeeper.quorum</name>
<value>xxx-zk1.cloudtable.com,xxx-zk2.cloudtable.com,xxx-zk3.cloudtable.com</value>
</property>
```

value is the domain name of the ZooKeeper cluster. Log in to the CloudTable console and choose Cluster Management. In the cluster list, locate the required cluster and obtain the address in the **Access Address (Intranet)** column.

----End

1.5.4.2 Creating the Configuration Object

Function Description

HBase obtains configuration items by loading a configuration file.

NOTE

1. Loading the configuration file is time-consuming. If unnecessary, use the same Configuration object.
2. Multi-thread synchronization is not considered in the sample code. If necessary, add it by yourself. Other sample codes are the same.

Sample Code

The following code snippets are in the **com.huawei.cloudtable.lemonIndex.examples.LemonIndexTestMain** packet.

```
private static void init() throws IOException {
  // Default load from conf directory
  conf = HBaseConfiguration.create(); // Note [1]
  String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
  Path hbaseSite = new Path(userdir + "hbase-site.xml");
  if (new File(hbaseSite.toString()).exists()) {
    conf.addResource(hbaseSite);
  }
}
```

Precautions

- Note [1] If the **conf** directory of the configuration file is added to the **classpath** path, the code for loading the specified configuration file can be skipped.

1.5.4.3 Enabling a Term Index When Creating a Table

Function Description

Follow instructions in **Creating a Table** to create a table. In addition, configure a term index schema in table properties.

Sample Code

```
public void testCreateTable() {
  LOG.info("Entering testCreateTable.");
  HTableDescriptor tableDesc = new HTableDescriptor(tableName);
  HColumnDescriptor cdm = new HColumnDescriptor(FAM_M);
  cdm.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
  tableDesc.addFamily(cdm);
  HColumnDescriptor cdn = new HColumnDescriptor(FAM_N);
  cdn.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
  tableDesc.addFamily(cdn);

  // Add bitmap index definitions.
  List<BitmapIndexDescriptor> bitmaps = new ArrayList<>();//(1)
  bitmaps.add(BitmapIndexDescriptor.builder()
    // Describe which column should be indexed.
    .setColumnName(FamilyOnlyName.valueOf(FAM_M))//(2)
    // Describe how to extract term(s) from KeyValue
    .setTermExtractor(TermExtractor.NAME_VALUE_EXTRACTOR)//(3)
    .build());
  // It will help to add several properties into HTableDescriptor.
  // SHARD_NUM should be less than the region number
  IndexHelper.enableAutoIndex(tableDesc, SHARD_NUM, bitmaps);//(4)

  List<byte[]> splitList = Arrays.stream(SPLIT.split(LemonConstants.COMMA))
    .map(s -> org.lemon.common.Bytes.toBytes(s.trim()))
    .collect(Collectors.toList());
  byte[][] splitArray = splitList.toArray(new byte[splitList.size()][]);

  Admin admin = null;
  try {
    // Instantiate an Admin object.
    admin = conn.getAdmin();
    if (!admin.tableExists(tableName)) {
      LOG.info("Creating table...");
      admin.createTable(tableDesc, splitArray);
      LOG.info(admin.getClusterStatus());
      LOG.info(admin.listNamespaceDescriptors());
      LOG.info("Table created successfully.");
    } else {
      LOG.warn("table already exists");
    }
  } catch (IOException e) {
    LOG.error("Create table failed.", e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting testCreateTable.");
}
```

Precautions

- (1) **BitmapIndexDescriptor** describes which fields use what rules to extract terms. One or more **BitmapIndexDescriptor** can be defined in a data table.
- (2) Defines which columns need to extract terms. The options are as follows:
  - **ExplicitColumnName**: Specifies a column.
  - **FamilyOnlyName**: Indicates all columns in a column family.
  - **PrefixColumnName**: Indicates columns with a specific prefix.
- (3) Defines a rule for extracting terms from columns. The options are as follows:
  - **QualifierExtractor**: Indicates that terms are extracted by column name.

    For example, if **qualifier** is **Male** and **value** is **1**, the extracted term is **Male**.
  - **QualifierValueExtractor**: Indicates that terms are extracted by column name and value.

    For example, if **qualifier** is **education** and **value** is **master**, the extracted term is **education:master**.
  - **QualifierArrayValueExtractor**: Indicates that multiple terms can be extracted. The value is in JSON array format.

    For example, if **qualifier** is **hobby** and **value** is **["basketball","football","volleyball"]**, the extracted terms are as follows:
    ```
    hobby:basketball
    hobby:football
    hobby:volleyball
    ```
  - **QualifierMapValueExtractor**: Indicates that multiple terms can be extracted. The value is in JSON map format.

    For example, if **qualifier** is **hobby** and **value** is **{"basketball":"9","football":"8","volleyball":"7"}**, the extracted terms are as follows:
    ```
    hobby:basketball
    hobby:football
    hobby:volleyball
    hobby:basketball_9
    hobby:football_8
    hobby:volleyball_7
    ```

- (4) The number of shards (**SHARD_NUM**) in the index table must be less than or equal to that in the data table.

1.5.4.4 Writing Data

The data write API is the same as the native HBase API.

Sample Code

```
public void testPut() {
  LOG.info("Entering testPut.");
  try(Table table = conn.getTable(tableName)) {
    List<Put> puts = new ArrayList<>();
    Arrays.stream(SPLIT.split(LemonConstants.COMMA))
      .forEach(startkey -> {
        // Instantiate a Put object.
        Put put = new Put(Bytes.toBytes(startkey + "-rowkey001"));
        put.addColumn(FAM_M, QUA_M, Bytes.toBytes("bachelor"));
        put.addColumn(FAM_N, QUA_N, Bytes.toBytes("xiaowang"));
        puts.add(put);

        Put put1 = new Put(Bytes.toBytes(startkey + "-rowkey0012"));
        put1.addColumn(FAM_M, QUA_M, Bytes.toBytes("master"));
        put1.addColumn(FAM_N, QUA_N, Bytes.toBytes("xiaoming"));
        puts.add(put1);

        // Submit a put request.
        try {
          table.put(puts);
        } catch (IOException e) {
          LOG.info("put exception", e);
        }
      });
    LOG.info("Put successfully.");
  } catch (IOException e) {
    LOG.error("Put failed ", e);
  }
  LOG.info("Exiting testPut.");
}
```

1.5.4.5 Common Query

Function Description

Based on the Lucene syntax, CloudTable term index provides the self-developed query API **LemonTable.query(LemonQuery query)**.

Sample Code

```
public void testNormalQuery() {
  LOG.info("Entering testNormalQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
    // Build LemonQuery.
    LemonQuery query = LemonQuery.builder()
      // Set ad-hoc query condition.
      .setQuery("education:bachelor OR education:master") //(1)
      // Set how many rows should be cached on client for the initial request.
      .setCaching(10) //(2)
      // Set return column family/columns.
      .addFamily(FAM_M)
      .addColumn(FAM_N, QUA_N) //(3)
      // Set return return result just contains rowkeys, no any qualifier
      // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
      //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
      .build();
    ResultSet resultSet = lemonTable.query(query);
    // Read result rows.

    int count = resultSet.getCount();
    LOG.info("the entity count of query is " + count);

    List<EntityEntry> entries = resultSet.listRows();
    for (EntityEntry entry : entries) {
      Map<String, Map<String, String>> fams = entry.getColumns();
      for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
        String family = familyEntry.getKey();
        Map<String, String> qualifiers = familyEntry.getValue();
        for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
          String Qua = qualifier.getKey();
          String value = qualifier.getValue();
          LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
            + family + ":" + Qua + ", value is " + value);
        }
      }
    }
  } catch (IOException e) {
    LOG.error("testNormalQuery failed ", e);
  }

  LOG.info("Exiting testNormalQuery.");
}
```

Precautions

(1) Indicates a query condition, which complies with the Lucene syntax/BNF normal form. Example:

```
"Male ANDMarried AND AGE:25-30 AND BLOOD_TYPE:A"
"Male ANDMarried AND (AGE:25-30 OR AGE:30-35)AND BLOOD_TYPE:A"
```

(2) Indicates the number of rows of data to be returned in the query result.

(3) Indicates which qualifiers are returned in each row of data. If only a column family is set, all columns in the column family are returned.

1.5.4.6 Sampling Query

Function Description

Set **setSampling()** in addition to the settings in **Common Query**. During query, a shard is randomly selected from an index table to execute a query job.

Sample Code

```
public void testSamplingQuery() {
  LOG.info("Entering testSamplingQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
    // Build LemonQuery.
    LemonQuery query = LemonQuery.builder()
      // Set ad-hoc query condition.
      .setQuery("education:bachelor OR education:master")
      // Set how many rows should be cached on client for the initial request.
      .setCaching(10)
      // sampling query will be select one random shard/region to query
      .setSampling()
      // Set return column family/columns.
      .addFamily(FAM_M)
      .addColumn(FAM_N, QUA_N)
      // Set return return result just contains rowkeys, no any qualifier
      // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
      //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
      .build();
    ResultSet resultSet = lemonTable.query(query);
    // Read result rows.
    int count = resultSet.getCount();
    LOG.info("the entity count of query is " + count);

    List<EntityEntry> entries = resultSet.listRows();
    for (EntityEntry entry : entries) {
      Map<String, Map<String, String>> fams = entry.getColumns();
      for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
        String family = familyEntry.getKey();
        Map<String, String> qualifiers = familyEntry.getValue();
        for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
          String Qua = qualifier.getKey();
          String value = qualifier.getValue();
          LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
            + family + ":" + Qua + ", value is " + value);
        }
      }
    }
  } catch (IOException e) {
    LOG.error("testSamplingQuery failed ", e);
  }

  LOG.info("Exiting testSamplingQuery.");
  LOG.info("");
}
public void testSamplingQuery() {
  LOG.info("Entering testSamplingQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
    // Build LemonQuery.
    LemonQuery query = LemonQuery.builder()
      // Set ad-hoc query condition.
      .setQuery("education:bachelor OR education:master")
      // Set how many rows should be cached on client for the initial request.
      .setCaching(10)
      // sampling query will be select one random shard/region to query
      .setSampling()
      // Set return column family/columns.
      .addFamily(FAM_M)
      .addColumn(FAM_N, QUA_N)
      // Set return return result just contains rowkeys, no any qualifier
      // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
      //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
      .build();
    ResultSet resultSet = lemonTable.query(query);
    // Read result rows.
    int count = resultSet.getCount();
    LOG.info("the entity count of query is " + count);

    List<EntityEntry> entries = resultSet.listRows();
    for (EntityEntry entry : entries) {
      Map<String, Map<String, String>> fams = entry.getColumns();
      for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
        String family = familyEntry.getKey();
        Map<String, String> qualifiers = familyEntry.getValue();
        for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
          String Qua = qualifier.getKey();
          String value = qualifier.getValue();
          LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
            + family + ":" + Qua + ", value is " + value);
        }
      }
    }
  } catch (IOException e) {
    LOG.error("testSamplingQuery failed ", e);
  }

  LOG.info("Exiting testSamplingQuery.");
  LOG.info("");
}
```

1.5.4.7 Paging Query

Function Description

Run the query API to return brief data information, and then call the **listRows** API to turn pages.

Sample Code

```
public void testPagingQuery() {
  LOG.info("Entering testPagingQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
    // Build LemonQuery.
    LemonQuery query = LemonQuery.builder()
      // Set ad-hoc query condition.
      .setQuery("education:bachelor OR education:master")
      // Set how many rows should be cached on client for the initial request.
      .setCaching(10)
      // Set return column family/columns.
      .addFamily(FAM_M)
      .addColumn(FAM_N, QUA_N)
      // Set return return result just contains rowkeys, no any qualifier
      // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
      //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
      .build();
    ResultSet resultSet = lemonTable.query(query);
    // Read result rows.
    int count = resultSet.getCount();
    LOG.info("the entity count of query is " + count);

    // Read result page by page, every page show 10 lines data
    int maxPage = 100;
    final int lineNumPerPage = 5;
    for (int i = 0; i < maxPage; i++) {
      int start = lineNumPerPage * i;
      List<EntityEntry> entries = resultSet.listRows(start, lineNumPerPage);
      if (entries == null || entries.size() == 0)
      {
        break;
      }

      LOG.info("page " + (i + 1) + " count is " + entries.size() + ", result is following:");
      for (EntityEntry entry : entries) {
        Map<String, Map<String, String>> fams = entry.getColumns();
        for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
          String family = familyEntry.getKey();
          Map<String, String> qualifiers = familyEntry.getValue();
          for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
            String Qua = qualifier.getKey();
            String value = qualifier.getValue();
            LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
              + family + ":" + Qua + ", value is " + value);
          }
        }
      }
    }
  } catch (IOException e) {
    LOG.error("testPagingQuery failed ", e);
  }

  LOG.info("Exiting testPagingQuery.");
}
```

1.5.4.8 Statistics Query

Function Description

The total number of entities that meet the query conditions is returned. The detailed information about the data is not returned. The **setCountOnly()** parameter is set in the code.

Sample Code

```
public void testCountOnlyQuery() {
  LOG.info("Entering testCountOnlyQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
    // Build LemonQuery.
    LemonQuery query = LemonQuery.builder()
      // Set ad-hoc query condition.
      .setQuery("education:bachelor OR education:master")
      // just return how many entities meet the query condition, without any rowkey/column
      .setCountOnly()
      .build();
    ResultSet resultSet = lemonTable.query(query);
    // Read result rows.
    int count = resultSet.getCount();
    LOG.info("the entity count of query is " + count);
  } catch (IOException e) {
    LOG.error("testCountOnlyQuery failed ", e);
  }

  LOG.info("Exiting testCountOnlyQuery.");
}
```

1.6 Commissioning Applications

1.6.1 Commissioning Applications on Windows

1.6.1.1 Compiling and Running Applications

Scenario

You can run applications in the Windows development environment after application code development is complete.

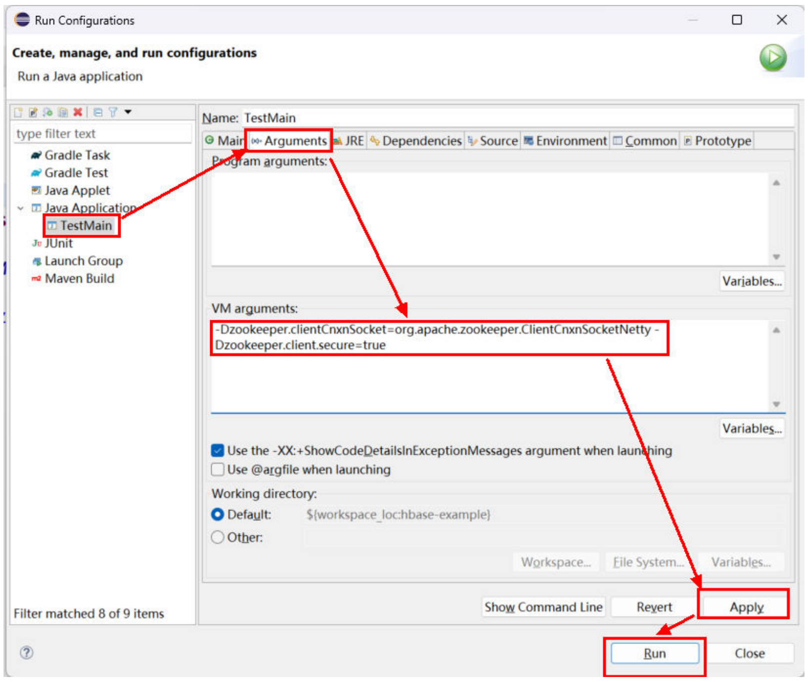
Procedure

- **HBase clusters with the encryption stream disabled**

  In the development environment (for example, Eclipse), right-click **TestMain.java** and choose **Run as** > **Java Application** from the shortcut menu to run the corresponding application project.

- **HBase clusters with the encryption stream enabled**

  In the development environment (for example, Eclipse), right-click **TestMain.java**, choose **Run as** > **Run Configurations**, add the environment variable "**-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty -Dzookeeper.client.secure=true**", and run the corresponding application project.



Name: TestMain

VM arguments:

```
-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty -
Dzookeeper.client.secure=true
```

[✓] Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching

[ ] Use @argfile when launching

Working directory:

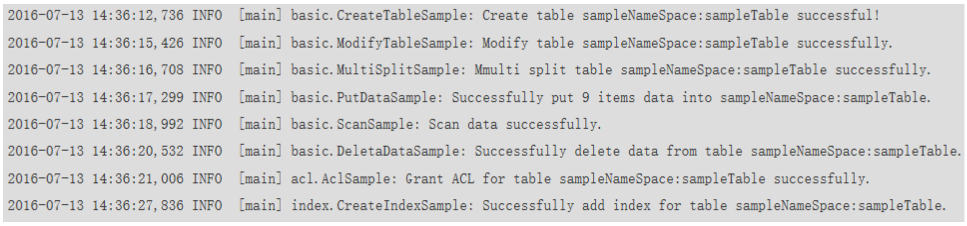
[✓ Default:] ${workspace_loc:hbase-example}

Other:

1.6.1.2 Viewing Commissioning Results

If no exception or failure information is displayed, the application running is successful.

Figure 1-8 Running succeeded



```
2016-07-13 14:36:12,736 INFO  [main] basic.CreateTableSample: Create table sampleNameSpace:sampleTable successful!
2016-07-13 14:36:15,426 INFO  [main] basic.ModifyTableSample: Modify table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:16,708 INFO  [main] basic.MultiSplitSample: Mmulti split table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:17,299 INFO  [main] basic.PutDataSample: Successfully put 9 items data into sampleNameSpace:sampleTable.
2016-07-13 14:36:18,992 INFO  [main] basic.ScanSample: Scan data successfully.
2016-07-13 14:36:20,532 INFO  [main] basic.DeletaDataSample: Successfully delete data from table sampleNameSpace:sampleTable.
2016-07-13 14:36:21,006 INFO  [main] acl.AclSample: Grant ACL for table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:27,836 INFO  [main] index.CreateIndexSample: Successfully add index for table sampleNameSpace:sampleTable.
```

NOTE

The following exception may occur when the sample code is running in the Windows OS, but it will not affect services.

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

Log description:

The log level is INFO by default and you can view more detailed information by changing the log level, such as DEBUG, INFO, WARN, ERROR, and FATAL. You can modify the **log4j.properties** file to change log levels, for example:

```
hbase.root.logger=INFO,console
log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=INFO
# Make these two classes DEBUG-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
```

1.6.2 Commissioning Applications on Linux

1.6.2.1 Compiling and Running an Application When a Client Is Installed

Scenario

HBase applications can run in a Linux environment where an HBase client is installed. After application code development is complete, you can upload a JAR file to the Linux environment to run applications.

Prerequisites

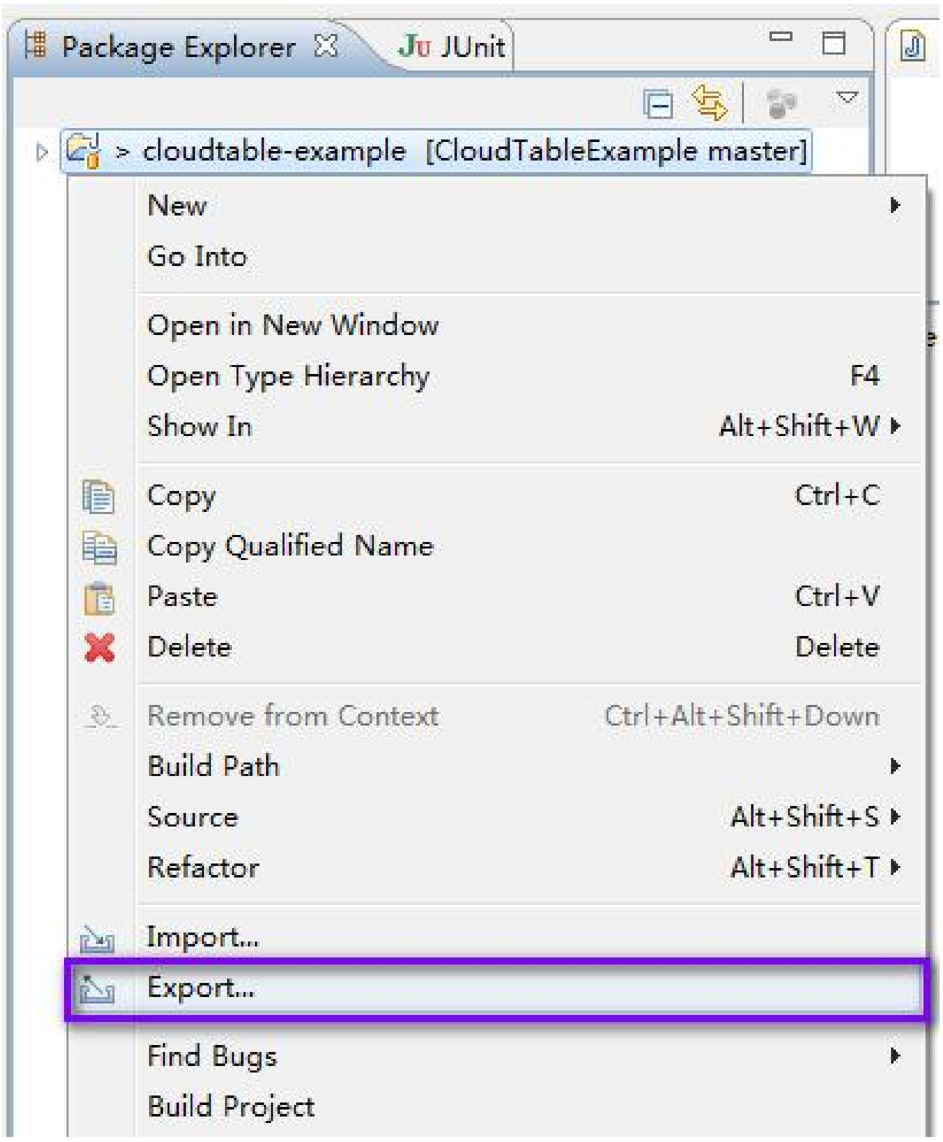
- You have installed an HBase client.
- You have installed a JDK in the Linux environment. The version of the JDK must be consistent with that of the JDK used by Eclipse to export the JAR file.

Procedure

Step 1 Export a JAR file.

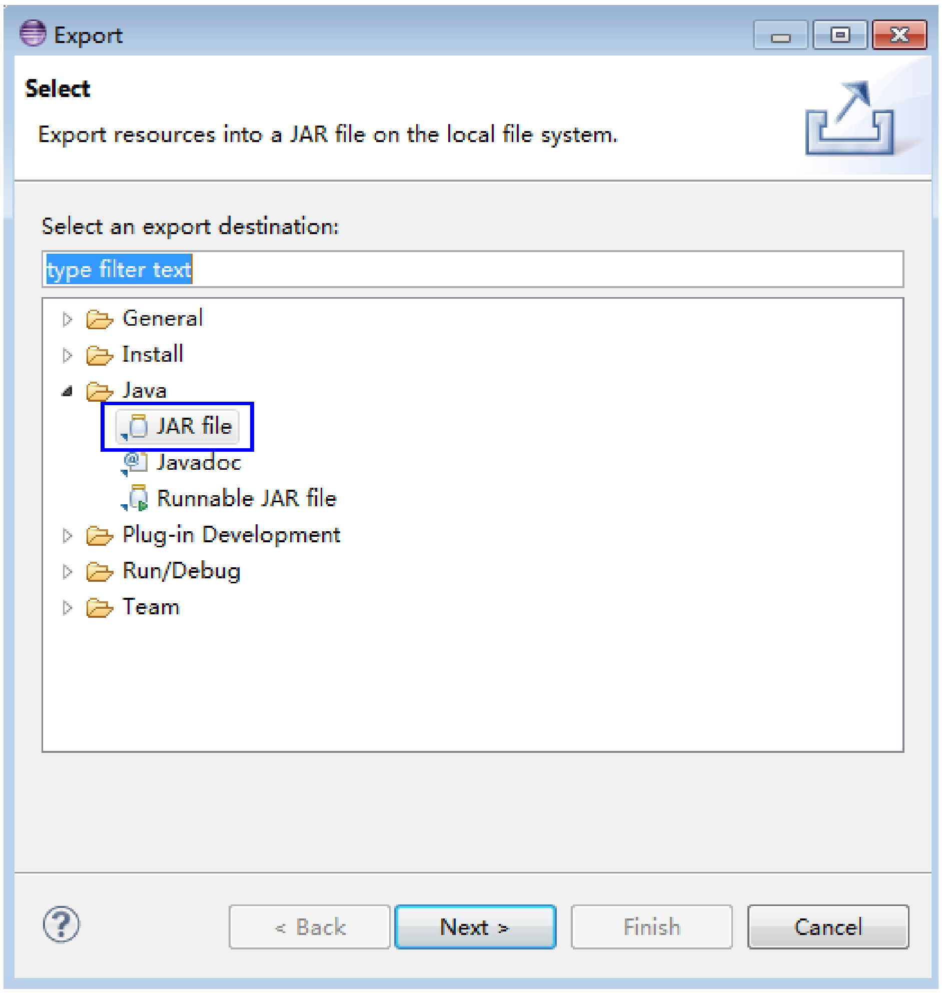
1. Right-click the sample project and choose **Export** from the shortcut menu.

Figure 1-9 Exporting a JAR file



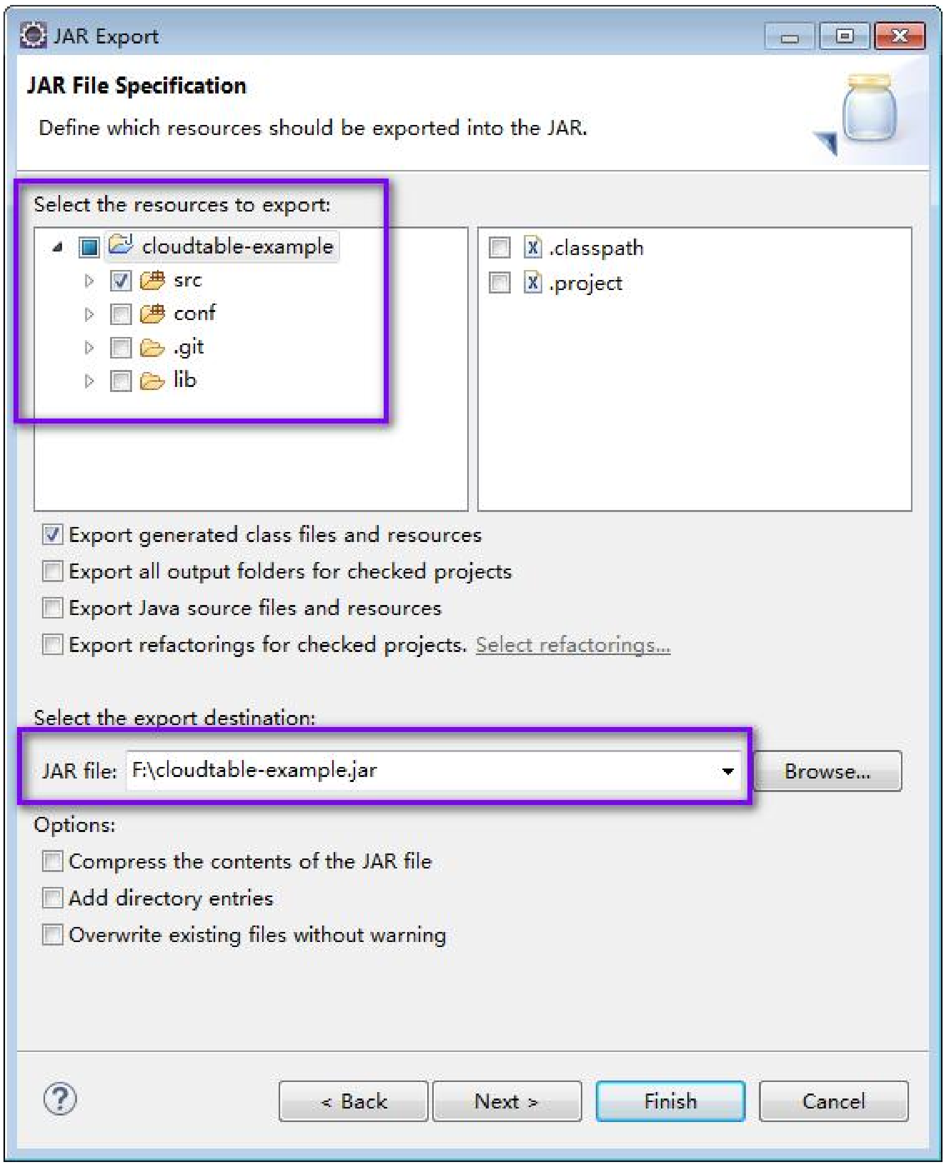
2. Select **JAR file** and click **Next**.

Figure 1-10 Selecting JAR file



3. Select the **src** and **conf** directories, and export the JAR file to the specified location. Click **Next** twice.

Figure 1-11 Selecting a path for exporting the JAR file



- [✓] Export generated class files and resources
- [ ] Export all output folders for checked projects
- [ ] Export Java source files and resources
- [ ] Export refactorings for checked projects.

JAR file: F:\cloudtable-example.jar

- [ ] Compress the contents of the JAR file
- [ ] Add directory entries
- [ ] Overwrite existing files without warning

4. Click **Finish**. Exporting the JAR file is complete.

Step 2 Run the JAR file.

1. Before running the JAR file on the Linux client, copy the JAR file generated in the application development environment to the **lib** directory in the client installation directory, and ensure that the permission of the JAR file is the same as that of other files.
2. Switch to the **bin** directory in the client directory as the installation user, and then run the following command to Run the JAR file:

```
[Ruby@cloudtable-08261700-hmaster-1-1 bin]# ./hbase
com.huawei.cloudtable.hbase.examples.TestMain
```

   com.huawei.cloudtable.hbase.examples.TestMain is used as an example. Use the actual code instead.

----End

1.6.2.2 Compiling and Running an Application When No Client Is Installed

Scenario

HBase applications can run in a Linux environment where an HBase client is not installed. After application code development is complete, you can upload a JAR file to the Linux environment to run applications.

Prerequisites

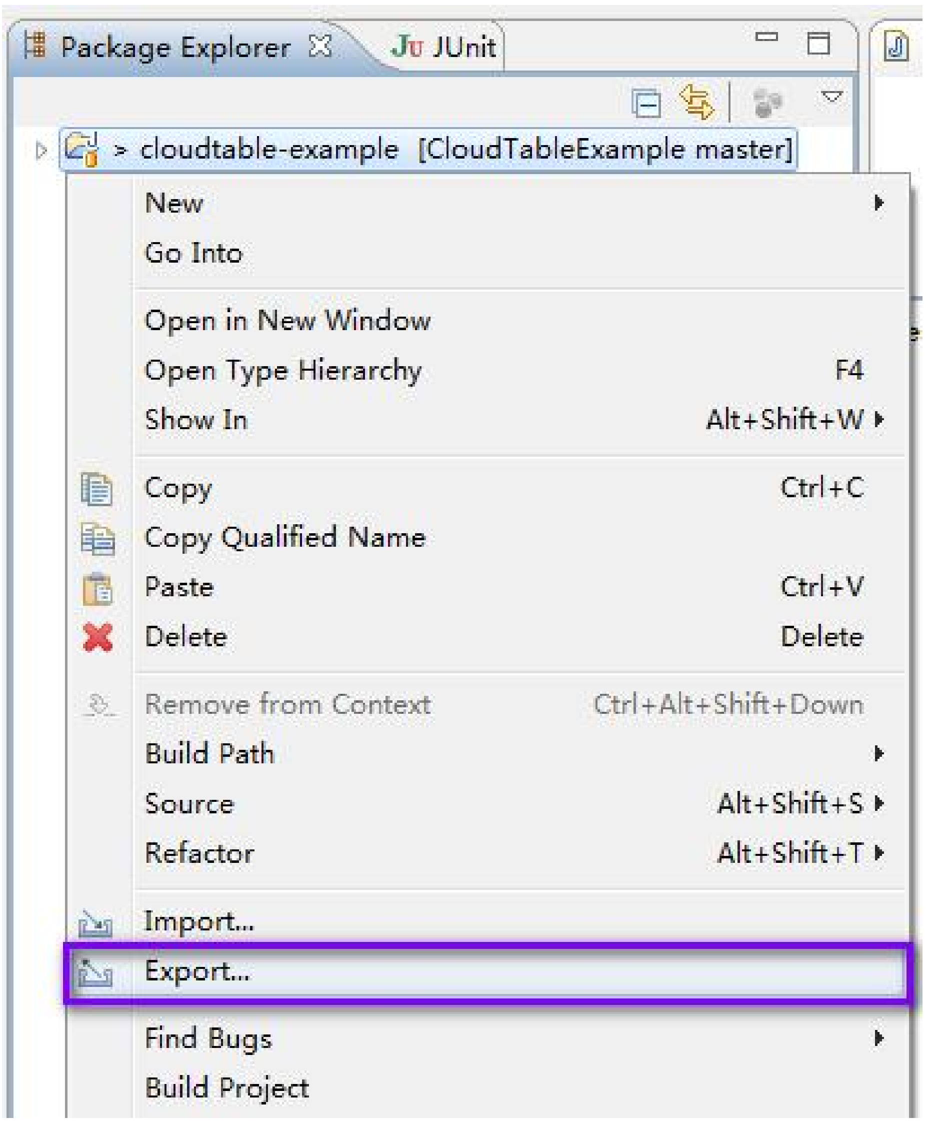
You have installed a JDK in the Linux environment. The version of the JDK must be consistent with that of the JDK used by Eclipse to export the JAR file.

Procedure

Step 1 Export a JAR file.

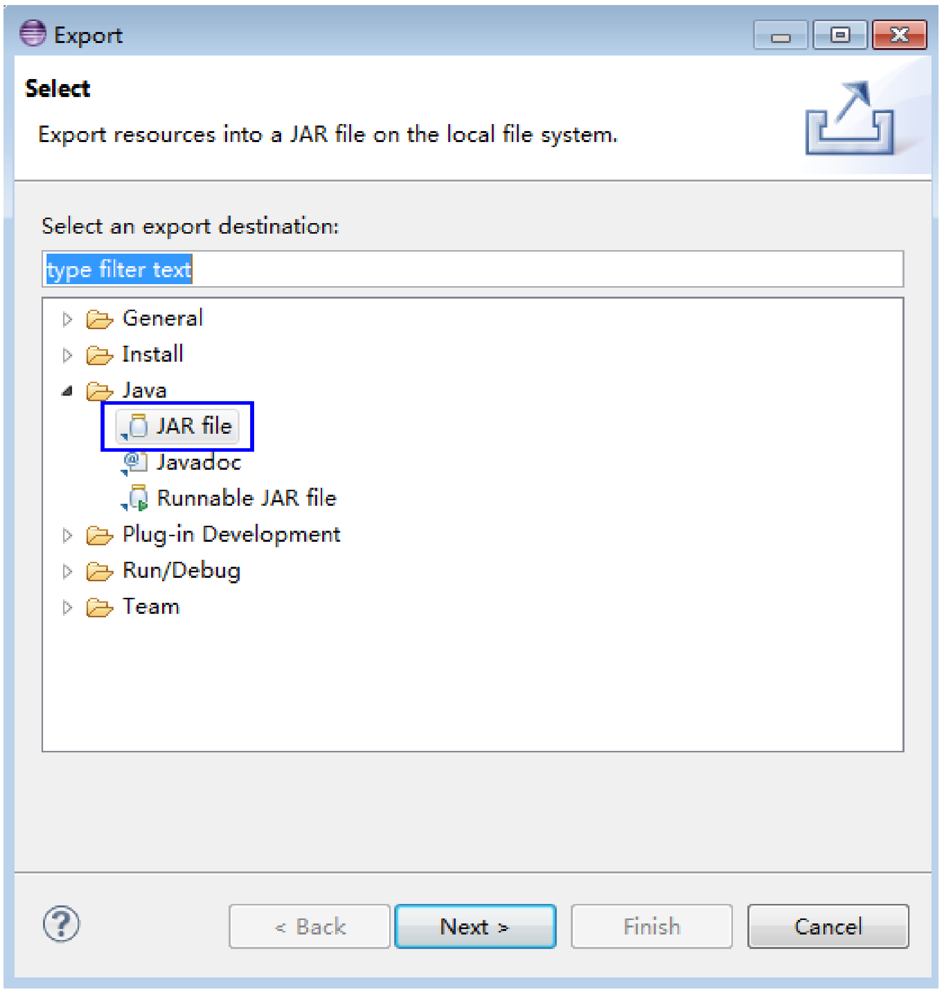
1. Right-click the sample project and choose **Export** from the shortcut menu.

Figure 1-12 Exporting a JAR file



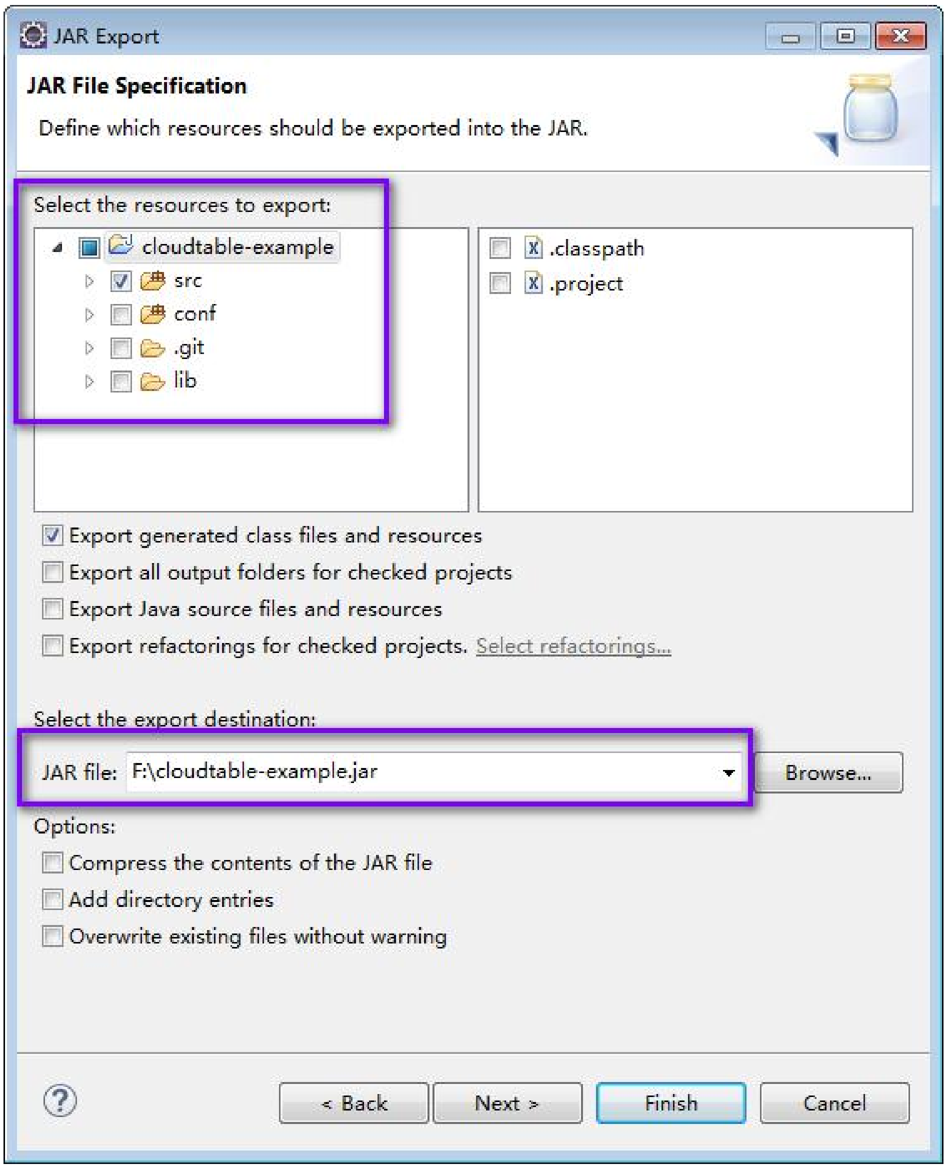
2. Select **JAR file** and click **Next**.

Figure 1-13 Selecting JAR file



3. Select the **src** directory, and export the JAR file to the specified location. Click **Next** twice.

Figure 1-14 Selecting a path for exporting the JAR file


JAR Export

JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

- cloudtable-example
  - [✓] src
  - [ ] conf
  - [ ] .git
  - [ ] lib

- [ ] .classpath
- [ ] .project

- [✓] Export generated class files and resources
- [ ] Export all output folders for checked projects
- [ ] Export Java source files and resources
- [ ] Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: F:\cloudtable-example.jar

Options:

- [ ] Compress the contents of the JAR file
- [ ] Add directory entries
- [ ] Overwrite existing files without warning

4. Click **Finish**. Exporting the JAR file is complete.

Step 2 Prepare the required JAR file and configuration file.

1. In the Linux environment, create a directory, for example, **/opt/test**, and create subdirectories **lib** and **conf**. Upload the JAR file in **lib** in the sample project and the JAR file exported in **Step 1** to the **lib** directory on Linux. Upload the configuration file in **conf** in the sample project to the **conf** directory on Linux.
2. In the **/opt/test** root directory, create the **run.sh** script, modify the following content, and save the file:

```
#!/bin/sh
BASEDIR=`pwd`
SECURE=""
if [ $# -eq 1 ]; then
  SECURE="-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty -
Dzookeeper.client.secure=true"
fi
cd ${BASEDIR}
for file in ${BASEDIR}/lib/*.jar
```

```
do
i_cp=$i_cp:$file
echo "$file"
done
for file in ${BASEDIR}/conf/*
do
i_cp=$i_cp:$file
done
java -cp .${i_cp} ${SECURE} com.huawei.cloudtable.hbase.examples.TestMain
```

Step 3 Go to **/opt/test** and run the following command to run the JAR file:

- **HBase clusters with the encryption stream disabled**

  sh run.sh

- **HBase clusters with the encryption stream enabled**

  sh run.sh secure

  NOTE

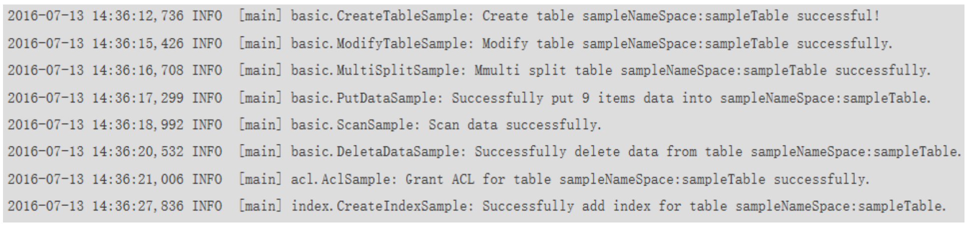
  If you use other methods to access an HBase cluster with the encryption stream enabled, you need to add the parameter "**-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty -Dzookeeper.client.secure=true**".

----End

1.6.2.3 Viewing Commissioning Results

If no exception or failure information is displayed, the application running is successful.

Figure 1-15 Running succeeded



```
2016-07-13 14:36:12,736 INFO  [main] basic.CreateTableSample: Create table sampleNameSpace:sampleTable successful!
2016-07-13 14:36:15,426 INFO  [main] basic.ModifyTableSample: Modify table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:16,708 INFO  [main] basic.MultiSplitSample: Mmulti split table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:17,299 INFO  [main] basic.PutDataSample: Successfully put 9 items data into sampleNameSpace:sampleTable.
2016-07-13 14:36:18,992 INFO  [main] basic.ScanSample: Scan data successfully.
2016-07-13 14:36:20,532 INFO  [main] basic.DeletaDataSample: Successfully delete data from table sampleNameSpace:sampleTable.
2016-07-13 14:36:21,006 INFO  [main] acl.AclSample: Grant ACL for table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:27,836 INFO  [main] index.CreateIndexSample: Successfully add index for table sampleNameSpace:sampleTable.
```

The log level is INFO by default and you can view more detailed information by changing the log level, such as DEBUG, INFO, WARN, ERROR, and FATAL. You can modify the **log4j.properties** file to change log levels, for example:

```
hbase.root.logger=INFO,console
log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=INFO
# Make these two classes DEBUG-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
```

1.7 External APIs

1.7.1 HBase Java API

HBase adopts the same APIs as those of Apache HBase. For details, visit **https://hbase.apache.org/1.2/apidocs/index.html**.

2 ClickHouse Application Development Guide

2.1 ClickHouse Table Engines

Background

The table engine determines:

- Where to write and read data.
- Which queries are supported.
- Whether concurrent data access is supported.
- Whether indexes are supported.
- Whether multi-thread requests can be executed.
- Parameters used for data replication.

This section describes MergeTree and Distributed engine families, which are the most important and frequently used.

Overview

A table engine is a table type. ClickHouse table engine determines how to store and read data, whether indexes are supported, and whether active/standby replication is supported. The following table lists ClickHouse table engines to help you get started with ClickHouse.

Table 2-1 Table engines

Engine Family	Description	Engine	Description
MergeTree	● MergeTree engines are the most universal and functional and are mainly used for heavy-load tasks. They support quick write of a large amount of data and subsequent data processing. ● MergeTree engines support data replication, partitioning, and data sampling.	MergeTree	● Data is stored by partition and block based on partitioning keys. ● Data index is sorted based on primary keys and the **ORDER BY** sorting keys. ● Data replication is supported by table engines prefixed with Replicated. ● Data sampling is supported. When data is written, a table with this type of engine divides data into different folders based on the partitioning key. Each column of data in the folder is an independent file. A file that records serialized index sorting is created. This structure reduces the volume of data to be retrieved during data reading, greatly improving query efficiency.
		RelacingMergeTree	This table engine removes duplicates that have the same primary key value. The MergeTree table engine does not support this feature.
		CollapsingMergeTree	CollapsingMergeTree defines a **Sign** field to record status of data rows. If **Sign** is **1**, the data in this row is valid. If **Sign** is **-1**, the data in this row needs to be deleted.
		VersionedCollapsingMergeTree	This table engine allows you to add the **Version** column to the **CREATE TABLE** statement. This helps resolve the issue that the CollapsingMergeTree table engine cannot collapse or delete rows as expected if the rows are inserted in an incorrect order.

Engine Family	Description	Engine	Description
		SummigMergeTree	This table engine pre-aggregates primary key columns and combines all rows that have the same primary key into one row. This helps reduce storage usage and improves aggregation performance.
		AggregatingMergeTree	This table engine is a pre-aggregation engine and is used to improve aggregation performance. When merging partitions, the AggregatingMergeTree engine aggregates data based on predefined conditions, calculates data based on predefined aggregate functions, and saves the data in binary format to tables.
		GraphiteMergeTree	This table engine is used to store and roll up Graphite data. This helps reduce storage space and makes Graphite data queries more efficient.
Replicated*MergeTree	All engines of the MergeTree family in ClickHouse prefixed with Replicated become MergeTree engines that support replicas.	Replicated*MergeTree series	Replicated series engines use ZooKeeper to synchronize data. When a replicated table is created, all replicas of the same shard are synchronized based on the information registered with ZooKeeper.
Distributed	-	Distributed	This table engine does not store data and performs distributed queries on multiple servers.

MergeTree

- Creating a table.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse cluster name]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
    ...
    INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,
    INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
) ENGINE = MergeTree()
ORDER BY expr
[PARTITION BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[TTL expr [DELETE|TO DISK 'xxx'|TO VOLUME 'xxx'], ...]
[SETTINGS name=value, ...]
```

- The following is an example.

```
CREATE TABLE default.test (name1 DateTime,name2 String,name3 String,name4 String,name5 Date)
ENGINE = MergeTree() PARTITION BY toYYYYMM(name5) ORDER BY (name1, name2) SETTINGS
index_granularity = 8192;
```

Parameters in the example are described as follows:

Table 2-2 Parameter description

Parameter	Description
ENGINE = MergeTree()	MergeTree table engine.
PARTITION BY toYYYYMM(name5)	Partition. The sample data is partitioned by month, and a folder is created for each month.
ORDER BY	Sorting field. A multi-field index can be sorted. If the first field is the same, the second field is used for sorting, and so on.
index_granularity = 8192	Granularity of a sorting index. One index value is recorded for every 8,192 data records.

NOTE

If the data to be queried exists in a partition or sorting field, the data query efficiency is greatly improved.

ReplacingMergeTree

ClickHouse provides the ReplacingMergeTree table engine to remove duplicates that have the same primary key value. The MergeTree table engine does not support this feature.

- Create a table.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse cluster name]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = ReplacingMergeTree([ver])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

CollapsingMergeTree

The CollapsingMergeTree table engine removes the limits of the ReplacingMergeTree table engine. This table engine allows you to add the **Sign** column to the **CREATE TABLE** statement. Rows are classified into two types. If **Sign** is **1**, the row is a "state" row and is used to add states. If **Sign** is **–1**, the row is a "cancel" row and is used to delete states.

- Statements for creating a table:
```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse cluster name]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = CollapsingMergeTree(sign)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```
- Example
  – Sample data

    Assume that you need to calculate how many pages users visited on a website and how long they were there. At a specific time point, write the following row with the state of the user's activity.

    Table 2-3 Sample data

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1
4324182021466249494	5	146	-1
4324182021466249494	6	185	1

    ▪ **Sign**: Name of the column with the type of row. **1** is a "state" row and **–1** is a "cancel" row.

  – Create the **Test** table.
```
CREATE TABLE Test(UserID UInt64,PageViews UInt8,Duration UInt8,Sign Int8)ENGINE =
CollapsingMergeTree(Sign) ORDER BY UserID;
```
  – Insert data.

    ▪ Insert data for the first time.
```
INSERT INTO Test VALUES (4324182021466249494, 5, 146, 1);
```
    ▪ Insert data for the second time.
```
INSERT INTO Test VALUES (4324182021466249494, 5, 146, -1),(4324182021466249494, 6,
185, 1);
```
  – View data.
```
SELECT * FROM Test;
```
    The following query result is returned:
```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         5 │      146 │   -1 │
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘

┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
```

```
│ 4324182021466249494 │         5 │      146 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

– Aggregate data in a specified column.

```
SELECT UserID,sum(PageViews * Sign) AS PageViews,sum(Duration * Sign) AS Duration FROM
Test GROUP BY UserID HAVING sum(Sign) > 0;
```

The command output is as follows:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┐
│ 4324182021466249494 │         6 │      185 │
└─────────────────────┴───────────┴──────────┘
```

– Perform force collapsing on data.

```
SELECT * FROM Test FINAL;
```

The command output is as follows:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
│ 4324182021466249494 │         6 │      185 │    1 │
└─────────────────────┴───────────┴──────────┴──────┘
```

VersionedCollapsingMergeTree

ClickHouse provides the VersionedCollapsingMergeTree table engine to resolve the issue that the CollapsingMergeTree table engine cannot collapse or delete rows as expected if the rows are inserted in an incorrect order. The VersionedCollapsingMergeTree table engine allows you to add the **Version** column to the **CREATE TABLE** statement to record the mapping between the "state" rows and "cancel" rows. During background compaction, rows with the same primary key, **Version**, and **Sign** are collapsed (deleted).

- Create a table.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse cluster name]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = VersionedCollapsingMergeTree(sign, version)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

- Example
  – Sample data

    Assume that you need to calculate how many pages users visited on a website and how long they were there. At a specific time point, write the following row with the state of the user's activity.

Table 2-4 Sample data

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

- **Sign**: Name of the column with the type of row. **1** is a "state" row and **–1** is a "cancel" row.
- **Version**: Name of the column with the version of the object state.

– Create the **T** table.

```
CREATE TABLE T(UserID UInt64,PageViews UInt8,Duration UInt8,Sign Int8,Version
UInt8)ENGINE = VersionedCollapsingMergeTree(Sign, Version)ORDER BY UserID;
```

– Insert two different parts of data.

```
INSERT INTO T VALUES (4324182021466249494, 5, 146, 1, 1);
INSERT INTO T VALUES (4324182021466249494, 5, 146, -1, 1),(4324182021466249494, 6, 185,
1, 2);
```

– View data.

```
SELECT * FROM T;
```

– Aggregate data in a specified column.

```
SELECT UserID, sum(PageViews * Sign) AS PageViews,sum(Duration * Sign) AS Duration,Version
FROM T GROUP BY UserID, Version HAVING sum(Sign) > 0;
```

The query result is as follows:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Version─┐
│ 4324182021466249494 │         6 │      185 │       2 │
└─────────────────────┴───────────┴──────────┴─────────┘
```

– Perform force collapsing on data.

```
SELECT * FROM T FINAL;
```

The query result is as follows:

```
┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
│ 4324182021466249494 │         6 │      185 │    1 │       2 │
└─────────────────────┴───────────┴──────────┴──────┴─────────┘
```

SummingMergeTree

The SummingMergeTree table engine pre-aggregates primary key columns and combines all rows that have the same primary key into one row. This helps reduce storage usage and improves aggregation performance.

- Create a table.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse cluster name]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = SummingMergeTree([columns])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

- Example

– Create a SummingMergeTree table named **testTable**.

```
CREATE TABLE testTable(id UInt32,value UInt32)ENGINE = SummingMergeTree() ORDER BY id;
```

– Inserts data to the **testTable** table.

```
INSERT INTO testTable Values(5,9),(5,3),(4,6),(1,2),(2,5),(1,4),(3,8);
INSERT INTO testTable Values(88,5),(5,5),(3,7),(3,5),(1,6),(2,6),(4,7),(4,6),(43,5),(5,9),(3,6);
```

– Query all data in unmerged parts.

```
SELECT * FROM testTable;
```

The following query result is returned:

```
┌─id─┬─value─┐
│  1 │     6 │
│  2 │     5 │
│  3 │     8 │
│  4 │     6 │
│  5 │    12 │
└────┴───────┘
┌─id─┬─value─┐
│  1 │     6 │
│  2 │     6 │
│  3 │    18 │
│  4 │    13 │
│  5 │    14 │
│ 43 │     5 │
│ 88 │     5 │
└────┴───────┘
```

– If ClickHouse has not summed up all rows and you need to aggregate data by ID, use the **sum** function and **GROUP BY** statement.

```
SELECT id, sum(value) FROM testTable GROUP BY id;
```

The following query result is returned:

```
┌─id─┬─sum(value)─┐
│  4 │         19 │
│  3 │         26 │
│ 88 │          5 │
│  2 │         11 │
│  5 │         26 │
│  1 │         12 │
│ 43 │          5 │
└────┴────────────┘
```

– Merge rows manually.

```
OPTIMIZE TABLE testTable;
```

Query data in the table.

```
SELECT * FROM testTable;
```

The following query result is returned:

```
┌─id─┬─value─┐
│  1 │    12 │
│  2 │    11 │
│  3 │    26 │
│  4 │    19 │
│  5 │    26 │
│ 43 │     5 │
│ 88 │     5 │
└────┴───────┘
```

NOTE

- SummingMergeTree uses the **ORDER BY** sorting keys as the condition keys to aggregate data. If sorting keys are the same, data records are merged into one and the specified merged fields are aggregated.
- Data is pre-aggregated only when merging is executed in the background, and the merging execution time cannot be predicted. Therefore, it is possible that some data has been pre-aggregated and some data has not been aggregated. Therefore, the **GROUP BY** statement must be used during aggregation.

AggregatingMergeTree

The AggregatingMergeTree table engine is also used for pre-aggregation and can improve the aggregation performance.

- Create a table.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse cluster name]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = AggregatingMergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[TTL expr]
[SETTINGS name=value, ...]
```

- Example

  You do not need to set the AggregatingMergeTree parameter separately. When partitions are merged, data in each partition is aggregated based on the **ORDER BY** sorting key. You can set the aggregate functions to be used and column fields to be calculated by defining the AggregateFunction type.

  – Create a table.

  ```
  create table test_table (name1 String,name2 String,name3
  AggregateFunction(uniq,String),name4 AggregateFunction(sum,Int),name5 DateTime) ENGINE
  = AggregatingMergeTree() PARTITION BY toYYYYMM(name5) ORDER BY (name1,name2)
  PRIMARY KEY name1;
  ```

  When data of the AggregateFunction type is written or queried, the ***state** and ***merge** functions need to be called. The asterisk (*) indicates the aggregate functions used for defining the field type. In the table creation example, the **uniq** and **sum** functions are specified for the **name3** and **name4** fields defined in the **test_table**, respectively. Therefore, you need to call the **uniqState** and **sumState** functions and run the **INSERT** and **SELECT** statements when writing data into the table.

  – Insert data.

  ```
  insert into test_table select '8','test1',uniqState('name1'),sumState(toInt32(100)),'2021-04-30
  17:18:00';
  insert into test_table select '8','test1',uniqState('name1'),sumState(toInt32(200)),'2021-04-30
  17:18:00';
  ```

  – Query the data.

  ```
  select name1,name2,uniqMerge(name3),sumMerge(name4) from test_table group by
  name1,name2;
  ```

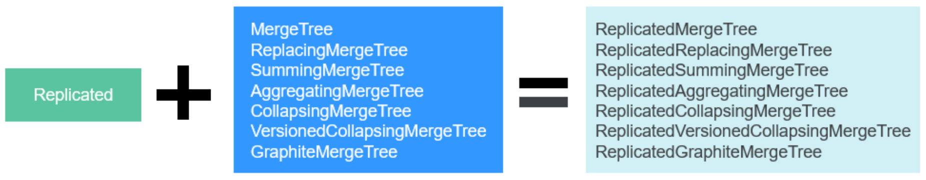
  The following query result is returned:

  ```
  ┌─name1─┬─name2─┬─uniqMerge(name3)─┬─sumMerge(name4)─┐
  │ 8     │ test1 │                1 │             300 │
  └───────┴───────┴──────────────────┴─────────────────┘
  ```

Replicated*MergeTree Engines

All engines of the MergeTree family in ClickHouse prefixed with Replicated become MergeTree engines that support replicas.

Figure 2-1 MergeTree table engines


Replicated
MergeTree
ReplacingMergeTree
SummingMergeTree
AggregatingMergeTree
CollapsingMergeTree
VersionedCollapsingMergeTree
GraphiteMergeTree
ReplicatedMergeTree
ReplicatedReplacingMergeTree
ReplicatedSummingMergeTree
ReplicatedAggregatingMergeTree
ReplicatedCollapsingMergeTree
ReplicatedVersionedCollapsingMergeTree
ReplicatedGraphiteMergeTree

- Template for creating a Replicated engine:

```
ENGINE = Replicated*MergeTree('ZooKeeper storage path','Replica name', ...)
```

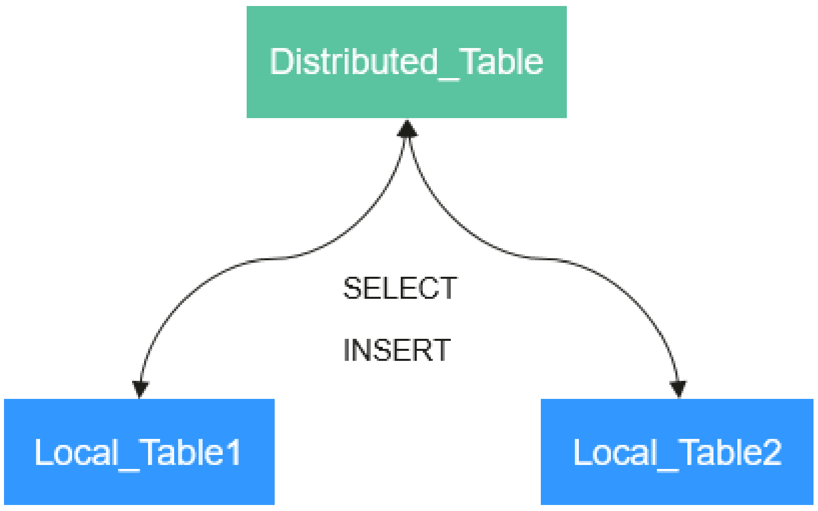
Table 2-5 Parameters

Parameter	Description
ZooKeeper storage path	Path for storing table data in ZooKeeper. The path format is **/clickhouse/tables/{shard}/** *Database name*/ *Table name*.
Replica name	**{replica}** is typically used to represent the replica name.

Distributed Table Engines

Tables with Distributed engine do not store any data of their own, but serve as a transparent proxy for data shards and can automatically transmit data to each node in the cluster. Distributed tables need to work with other local data tables. Distributed tables distribute received read and write tasks to each local table where data is stored.

Figure 2-2 Distributed



- Template for creating a distributed engine:

```
ENGINE = Distributed(cluster_name, database_name, table_name, [sharding_key])
```

Table 2-6 Distributed parameters

Parameter	Description
cluster_name	Cluster name. When a distributed table is read or written, the cluster configuration information is used to search for the corresponding ClickHouse instance node.
database_name	Database name.
table_name	Name of a local table in the database. It is used to map a distributed table to a local table.
sharding_key	Sharding key, based on which a distributed table distributes data to each local table.

- Example
  - Create a local ReplicatedMergeTree table named **demo**.
    ```
    CREATE TABLE default.demo ON CLUSTER default_cluster( `EventDate` DateTime, `id`
    UInt64)ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/demo', '{replica}')
    PARTITION BY toYYYYMM(EventDate) ORDER BY id;
    ```
  - Create a Distributed table named **demo_all** based on the local table **demo**.
    ```
    CREATE TABLE default.demo_all ON CLUSTER default_cluster( `EventDate` DateTime, `id`
    UInt64)ENGINE = Distributed(default_cluster, default, demo, rand());
    ```
- Rules for creating a distributed table:
  - When creating a distributed table, add **ON CLUSTER** *cluster_name* to the table creation statement so that the statement can be executed once on a ClickHouse instance and then distributed to all instances in the cluster for execution.
  - Generally, a distributed table is named in the following format: *Local table name*_all. It forms a one-to-many mapping with local tables. Then, multiple local tables can be operated using the distributed table proxy.
  - Ensure that the structure of a distributed table is the same as that of local tables. If they are inconsistent, no error is reported during table creation, but an exception may be reported during data query or insertion.

2.2 SQL Syntax

2.2.1 Data Types

This section describes the data types that are supported by ClickHouse.

Supported Data Types

Table 2-7 Supported data types

Type	Keyword	Data Type	Description
Integer	Int8	Int8	Value range: [–128, 127]
	Int16	Int16	Value range: [–32768, 32767]
	Int32	Int32	Value range: [–2147483648, 2147483647]
	Int64	Int64	Value range: [–9223372036854775808, 9223372036854775807]
Floating point	Float32	Single-precision floating point	Similar to the Float type in the C programming language. A single-precision floating point number occupies four bytes in storage of a computer and is represented in 32-bit binary.
	Float64	Double-precision floating point	Similar to the Double type in the C programming language. A double-precision floating point number occupies eight bytes in storage of a computer and is represented in 64-bit binary.
Decimal	Decimal	Decimal	A signed fixed-point number that can ensure precision during addition, subtraction, and multiplication operations. Decimal values can be in the following formats: ● Decimal(P, S) ● Decimal32(S) ● Decimal64(S) ● Decimal128(S) **NOTE** P stands for precision. The valid range is [1:38]. It determines the number of decimal digits (including fractions) that can be contained. S stands for scale. The valid range is [0:P]. It determines the number of decimal places of a number.

Type	Keyword	Data Type	Description
String	String	String	A string can be of a random length. It can contain any set of bytes, including null bytes. Therefore, the String type can replace the VARCHAR, BLOB, and CLOB types in other database management systems.
	FixedString	Fixed-length string	When the length of the data happens to be N bytes, using the FixedString type is more efficient than other types. In other cases, efficiency may be impaired. The following values can be effectively stored in columns of the FixedString type: ● IP addresses represented in binary (FixedString (16) for IPv6) ● Language codes (ru_RU, en_US...) ● Currency codes (USD, RUB...) ● Binary representation of hash values (FixedString (16) for MD5 and FixedString (32) for SHA256)
Date and time	Date	Date	A Date value takes up two bytes, indicating the date value from 1970-01-01 (unsigned) to the current time. Date values are stored without the time zone.
	DateTime	Timestamp	A Unix timestamp value takes up four bytes (unsigned). Value range of this type is the same as the Date type. The minimum value is 1970-01-01 00:00:00. Timestamp values are accurate to seconds. Leap seconds are not supported. The system time zone will be used when the client or server is started.
	DateTime64	DateTime64	This type allows you to store both the date and time of a specific point in time.

Type	Keyword	Data Type	Description
Boolean	Boolean	Boolean	ClickHouse does not support the Boolean type. You can use the UInt8 type for Boolean values. Valid values are 0 and 1.
Array	Array	Array	An Array value is a collection of elements of the same data type. The elements can be of a random data type, even the Array type itself. However, multi-dimensional arrays are not recommended, because ClickHouse supports multi-dimensional arrays only to a limited extent. For example, you cannot store multi-dimensional arrays in MergeTree tables.
Tuple	Tuple	Tuple	A Tuple value is a collection of elements of different data types. Tuple values cannot be stored in tables, except for memory tables. You can use Tuple values to group temporary columns. In queries, you can use **IN** expressions and **lambda** functions with specific parameters to group temporary columns.

Type	Keyword	Data Type	Description
Domains	Domains	Domains	The implementation of the Domains type varies based on different values: ● If the values are IPv4 addresses, the Domains type is binary compatible with the UInt32 type. Compared with the UInt32 type, the Domains type saves the binary storage space and supports more readable input and output formats. ● If the values are IPv6 addresses, the Domains type is binary compatible with the FixedString (16) type. Compared with the FixedString (16) type, the Domains type saves the binary storage space and supports more readable input and output formats.
Enumeration	Enum8	Enum8	Value range: [–128, 127] An Enum value stores the mapping of 'string'= integer, for example, **Enum8('hello' = 1, 'world' = 2)**.
	Enum16	Enum16	Value range: [–32768, 32767]
Nullable	Nullable	Nullable	Unless otherwise stated in ClickHouse server configurations, the default value of the NULLABLE type is NULL. Nullable values cannot be included in table indexes. Nullable values can be stored together with the normal values of TypeName. For example, columns of the Nullable(Int8) type can store values of the Int8 type, while rows without values store NULL.

Type	Keyword	Data Type	Description
Nested	nested	nested	A nested data structure is similar to a table inside a cell. You can specify the parameters of a nested data structure, such as field name and data type, the same way that you specify parameters in a **CREATE TABLE** statement. Each row in a **CREATE TABLE** statement can correspond to a random number of rows in a nested data structure. Example: **Nested (Name1 Type1,Name2 Type2, ...)**

2.2.2 CREATE DATABASE

This section describes the syntax of a **CREATE DATABASE** statement that is used to create a database in ClickHouse.

CREATE DATABASE Syntax

```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER ClickHouse cluster name];
```

Table 2-8 Parameter description

Parameter	Description
db_name	Database name.
IF NOT EXISTS	You can add the **IF NOT EXISTS** keyword phrase to a **CREATE DATABASE** statement. In this case, if the name specified in the statement has been used by an existing database, no database is created and no error is returned.
ON CLUSTER *ClickHouse cluster name*	This parameter specifies the name of a cluster.

NOTE

You can run the following statement to obtain the cluster name from the **cluster** field:

```
select cluster,shard_num,replica_num,host_name from system.clusters;
```

Use Example

- Create a database named **demo**.
```
create database demo ON CLUSTER default_cluster;
```

- View the created database.

```
host-172-16-30-9 :) show databases;
SHOW DATABASES
Query id: ced1af23-0286-40cc-9c7a-ccbca41178d8
┌─name───────────────┐
│ INFORMATION_SCHEMA │
│ default            │
│ demo               │
│ information_schema │
│ system             │
└────────────────────┘
5 rows in set. Elapsed: 0.002 sec.
```

2.2.3 CREATE TABLE

This section describes how to create a table in ClickHouse.

Creating a Local Table

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER ClickHouse cluster name]
(
name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
name2[type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
...
) ENGINE = engine_name()
[PARTITION BY expr_list]
[ORDER BY expr_list]
```

Table 2-9 Parameter description

Parameter	Description
database_name	Name of the database. The default value is the selected database.
table_name	Name of the local table.
ON CLUSTER *ClickHouse cluster name*	This parameter specifies that a local table is created on each node.. The parameter format is **ON CLUSTER** *ClickHouse cluster name*.
name1,name2	Column name.
ENGINE = engine_name()	Table engine type. When you create a table in a cluster of the double-replica edition, you must use the Replicated * engine that supports data replication among the engines in the MergeTree family. If you do not use the Replicated * engine, data cannot be replicated between replicas and inconsistent data query results are returned. When you use this engine to create a table, use one of the following methods to configure the parameters: ● ReplicatedMergeTree('/clickhouse/tables/{database}/{table}/{shard}', '{replica}'). The characters in the parameters cannot be changed. ● ReplicatedMergeTree(), which is equivalent to ReplicatedMergeTree('/clickhouse/tables/{database}/{table}/{shard}', '{replica}').

Parameter	Description
ORDER BY expr_list	Sort key. This parameter is required. The value can be a Tuple of a set of columns or an expression.
[PARTITION BY expr_list]	Partition key. In most cases, data is partitioned by date. You can specify another field or field expression as the partition key.

Examples:

- Create a database. For details, see **CREATE DATABASE**.
- Use the database.
```
use demo;
```
- Create a table named **demo.test**.
```
CREATE TABLE demo.test ON CLUSTER default_cluster(`EventDate` DateTime, `id` UInt64)ENGINE =
ReplicatedMergeTree('/clickhouse/tables/{shard}/default/test', '{replica}') PARTITION BY
toYYYYMM(EventDate) ORDER BY id;
```

Creating a Table by Copying the Structure of an Existing Table

Run the following statements to create a table by copying the schema of an existing table so that the table has the same structure as the source table.

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name2 ON CLUSTER ClickHouse cluster name
 AS [db.]table_name1 [ENGINE = engine_name];
```

Table 2-10 Parameter description

Parameter	Description
db	Name of the database. The default value is the selected database.
table_name1	Name of the source table from which the structure is copied.
table_name2	Name of the table that you want to create.
ON CLUSTER *ClickHouse cluster name*	This parameter specifies that a table is created on each node.. The parameter format is **ON CLUSTER** *ClickHouse cluster name*.
[ENGINE = engine_name]	Table engine type. If you do not specify a table engine when you create a table, the table engine of the source table is used by default.

Examples:

- Create a database.
```
create database demo;
```
- Use the database.
```
use demo;
```

- Create a table.
  ```
  create table demo_t(uid Int32,name String,age UInt32,gender String)engine = TinyLog;
  ```
- Copy the table structure.
  ```
  create table demo_t2 as demo_t;
  ```
- **Basic Syntax**

Creating a Table by Specifying a SELECT Clause in a CREATE TABLE Statement

You can use a specified table engine to create a table that has the same schema as the query result of the **SELECT** clause. The query result of the **SELECT** clause is populated to the table.

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name ENGINE = engine_name AS SELECT ...
```

Table 2-11 Parameter description

Parameter	Description
database_name	Name of the database. The default value is the selected database.
table_name	Table created using the **SELECT** statement.
ENGINE = engine_name()	Table engine type.
SELECT ...	**SELECT** clause.

Examples:

- Create a table.
  ```
  CREATE TABLE default.demo1 ON CLUSTER default_cluster( `EventDate` DateTime, `id` 
  UInt64)ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/demo1', '{replica}') 
  PARTITION BY toYYYYMM(EventDate) ORDER BY id;
  ```
- Run the **SELECT** statement to create a table.
  ```
  create table t3 ON CLUSTER default_cluster ENGINE =MergeTree() order by EventDate as select * 
  from default.demo1;
  ```
- Query the structures of tables **demo1** and **t3**.
  ```
  desc demo1;
  ```
  The query result shows that the structures of the two tables are the same.
  ```
  cloudtable-wlr-click-20230730-06-server-1-1 :) desc demo1;
  DESCRIBE TABLE demo1
  Query id: 712f6b91-668d-4f70-b160-aac8e52f63a4
  ┌─name──────┬─type─────┬─default_type─┬─default_expression─┬─comment─┬─codec_expre
  ssion─┬─ttl_expression─┐
  │ EventDate │ DateTime │              │                    │         │             │                │
  │ id        │ UInt64   │              │                    │         │             │                │
  └───────────┴──────────┴──────────────┴────────────────────┴─────────┴─────────────
  ──────┴────────────────┘
  2 rows in set. Elapsed: 0.001 sec.
  cloudtable-wlr-click-20230730-06-server-1-1 :) desc t3;
  DESCRIBE TABLE t3
  Query id: 11b67532-26f0-49c5-b36d-439d45c279bf
  ┌─name──────┬─type─────┬─default_type─┬─default_expression─┬─comment─┬─codec_expre
  ssion─┬─ttl_expression─┐
  │ EventDate │ DateTime │              │                    │         │             │                │
  ```

```
│ id       │ UInt64 │          │                    │        │                  │              │                │
└──────────┴────────┴──────────┴────────────────────┴────────┴──────────────────┴──────────────┴────────────────┘
2 rows in set. Elapsed: 0.001 sec.
```

2.2.4 DESC: Querying a Table Structure

This section describes the basic syntax and usage of the SQL statement for querying a table structure in ClickHouse.

Basic Syntax

```
DESC|DESCRIBE TABLE [database_name.]table [INTO OUTFILE filename] [FORMAT format]
```

Query the structures of tables **demo_t** and **demo_2** by referring to **Creating a Table by Copying the Structure of an Existing Table**.

```
cloudtable-wlr-click-20230730-06-server-1-1 :) desc demo_t;
DESCRIBE TABLE demo_t
Query id: 27a38d90-9459-430f-962e-881817789fc9
┌─name───┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_
expression─┐
│ uid    │ Int32  │              │                    │         │                  │                │
│ name   │ String │              │                    │         │                  │                │
│ age    │ UInt32 │              │                    │         │                  │                │
│ gender │ String │              │                    │         │                  │                │
└────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴────────────────┘
4 rows in set. Elapsed: 0.001 sec.
cloudtable-wlr-click-20230730-06-server-1-1 :) desc demo_t2;
DESCRIBE TABLE demo_t2
Query id: 60054fe3-794c-410a-be13-cd0b204a9129
┌─name───┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_
expression─┐
│ uid    │ Int32  │              │                    │         │                  │                │
│ name   │ String │              │                    │         │                  │                │
│ age    │ UInt32 │              │                    │         │                  │                │
│ gender │ String │              │                    │         │                  │                │
└────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴────────────────┘
4 rows in set. Elapsed: 0.001 sec.
```

2.2.5 CREATE VIEW

This section describes how to create a normal view in ClickHouse.

Creating a View

```
CREATE VIEW [IF NOT EXISTS] [db.]view_name [ON CLUSTER ClickHouse cluster name] AS SELECT ...
```

Table 2-12 Parameter description

Parameter	Description
db	Name of the database. The default value is the selected database.
view_name	View name.
[ON CLUSTER *ClickHouse cluster name*]	This parameter specifies that a view is created on each node. The parameter format is **ON CLUSTER** *ClickHouse cluster name*.

Parameter	Description
SELECT ...	**SELECT** clause. When you insert data into the source table that is specified in the **SELECT** clause in the view, the inserted data is transformed by the **SELECT** query and the final result is inserted into the view.

Examples:

1. Create a source table.
```
create table DB.table1 ON CLUSTER default_cluster (id Int16,name String) ENGINE = MergeTree()
ORDER BY (id);
```
2. Create a view.
```
CREATE VIEW test_view ON CLUSTER default_cluster AS SELECT * FROM DB.table1;
```
3. Insert data to the source table.
```
insert into DB.table1 values(1,'X'),(2,'Y'),(3,'Z');
```
4. Query a view.
```
SELECT * FROM test_view;
```
5. Delete a view.
```
drop table test_view ON CLUSTER default_cluster;
```

NOTE

- If the table creation statement contains **ON CLUSTER** *ClickHouse cluster name*, run the following command to delete the table:
```
drop table Table name ON CLUSTER default_cluster;
```
- If the table creation statement does not contain **ON CLUSTER** *ClickHouse cluster name*, run the following command to delete the table:
```
drop table Table name;
```

2.2.6 CREATE MATERIALIZED VIEW

This topic describes how to create a materialized view in ClickHouse.

Creating a Materialized View

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [db.]Materialized_name [TO[db.]name] [ON
CLUSTERClickHouse cluster name]
ENGINE = engine_name()
ORDER BY expr
[POPULATE]
AS SELECT ...
```

Table 2-13 Parameter description

Parameter	Description
db	Name of the database. The default value is the selected database.
Materialized_name	Name of the materialized view.
TO[db.]name	This parameter specifies that the data of the materialized view is inserted into a new table.

Parameter	Description
[ON CLUSTER *ClickHouse cluster name*]	This parameter specifies that a materialized view is created on each node. The parameter format is **ON CLUSTER** *ClickHouse cluster name*.
ENGINE = engine_name()	Table engine type.
[POPULATE]	**POPULATE** keyword. If you specify the **POPULATE** keyword when you create a materialized view, the data in the source table that is specified in the **SELECT** clause is inserted into the materialized view when the materialized view is being created. Otherwise, the materialized view contains only the data that is inserted into the source table after the materialized view is created. **NOTE** The **POPULATE** keyword is not recommended because the data that is written to the source table when the materialized view is being created is not inserted into the materialized view.
SELECT ...	**SELECT** clause. When you insert data into the source table that is specified in the **SELECT** clause in the materialized view, the inserted data is transformed by the **SELECT** query and the final result is inserted into the materialized view. **NOTE** A **SELECT** query can contain **DISTINCT**, **GROUP BY**, **ORDER BY**, and **LIMIT**. The corresponding transformations are performed independently on each block of the data that is inserted.

Examples:

1. Create a source table.
```
create table DB.table1 ON CLUSTER default_cluster (id Int16,name String) ENGINE = MergeTree()
ORDER BY (id);
```
2. Insert data.
```
insert into DB.table1 values(1,'X'),(2,'Y'),(3,'Z');
```
3. Create a materialized view based on the source table.
```
CREATE MATERIALIZED VIEW demo_view ON CLUSTER default_cluster ENGINE = MergeTree() ORDER
BY (id) AS SELECT * FROM DB.table1;
```
4. Query the materialized view.
```
SELECT * FROM demo_view;
```
NOTE

If the query result is empty, the data that is written to the source table before the materialized view is created cannot be queried if the **POPULATE** keyword is not specified.

5. Insert data to the **DB.table1** table.
```
insert into demo_view values(4,'x'),(5,'y'),(6,'z');
```
6. Query the materialized view.
```
SELECT * FROM demo_view;
```
The following query result is returned:
```
┌─id─┬─name─┐
│  4 │ x    │
│ 5 │ y  │
│ 6 │ z  │
└───┴────┘
```

2.2.7 INSERT INTO

This section describes how to insert data.

Basic Syntax

- The following code shows the syntax of an **INSERT INTO** statement:

```
INSERT INTO [db.]table [(c1, c2, c3)] VALUES (v11, v12, v13), (v21, v22, v23), ...
```

NOTE

For a field that is defined in the table structure but not specified in an **INSERT INTO** statement, the system fills in the field based on the following rules:

  - If an expression is defined for the default values of the field, the system calculates the default values by using this expression and then inserts the default values into the field.
  - If no expression is defined for the default values of the field, the system inserts **0** or empty strings into the field.

Insert data by referring to **Creating a Table by Copying the Structure of an Existing Table**.

```
insert into demo_t values(1,'Candy','23','M'),(2,'cici','33','F');
```

- Insert data by using the **SELECT** result.

```
INSERT INTO [db.]table [(c1, c2, c3)] SELECT ...
```

NOTE

Data in the fields that are queried in the **SELECT** clause are inserted into the fields that are specified in the **INSERT INTO** statement in strict order. The names of the two sets of fields do not have to be the same. The system converts the data types of the fields as needed.

Except for the VALUES type, all other data types do not support expressions such as now() and 1 + 2. The VALUES type allows you to use such expressions. However, these expressions are not recommended because the execution of these expressions is inefficient.

2.2.8 SELETC

This section describes how to execute a **SELECT** statement to query data.

Basic Syntax

```
SELECT [DISTINCT] expr_list
[FROM [database_name.]table | (subquery) | table_function] [FINAL]
[SAMPLE sample_coeff]
[ARRAY JOIN ...]
[GLOBAL] [ANY|ALL|ASOF] [INNER|LEFT|RIGHT|FULL|CROSS] [OUTER|SEMI|ANTI] JOIN (subquery)|table
(ON <expr_list>)|(USING <column_list>)
[PREWHERE expr]
[WHERE expr]
[GROUP BY expr_list] [WITH TOTALS]
[HAVING expr]
[ORDER BY expr_list] [WITH FILL] [FROM expr] [TO expr] [STEP expr]
[LIMIT [offset_value, ]n BY columns]
[LIMIT [n, ]m] [WITH TIES]
[UNION ALL ...]
[INTO OUTFILE filename]
[FORMAT format]
```

Examples:

- View the ClickHouse cluster information.
  ```
  select * from system.clusters;
  ```
- View the macros set for the node.
  ```
  select * from system.macros;
  ```
- Check the database capacity.
  ```
  select sum(rows) as "Total number of rows", formatReadableSize(sum(data_uncompressed_bytes)) as
  "Original size", formatReadableSize(sum(data_compressed_bytes)) as "Compression size",
  round(sum(data_compressed_bytes) / sum(data_uncompressed_bytes) * 100, 0) "Compression rate"
  from system.parts;
  ```
- Query the capacity of the **test** table. Add and modify the **WHERE** clause based on the site requirements.
  ```
  select sum(rows) as "Total number of rows", formatReadableSize(sum(data_uncompressed_bytes)) as
  "Original size", formatReadableSize(sum(data_compressed_bytes)) as "Compression size",
  round(sum(data_compressed_bytes) / sum(data_uncompressed_bytes) * 100, 0) "Compression rate"
  from system.parts where table in ('test') and partition like '2020-11-%' group by table;
  ```

2.2.9 ALTER TABLE

This section describes the basic syntax and usage of the SQL statement for modifying a table structure in ClickHouse.

Basic Syntax

```
ALTER TABLE [database_name].name [ON CLUSTER ClickHouse cluster name] ADD|DROP|CLEAR|
COMMENT|MODIFY COLUMN ...
```

NOTE

ALTER supports only *MergeTree, Merge, and Distributed engine tables.

Examples:

1. Create a table named **DB_table1**.
   ```
   CREATE TABLE DB_table1 ON CLUSTER default_cluster(Year UInt16,Quarter UInt8,Month
   UInt8,DayofMonth UInt8,DayOfWeek UInt8,FlightDate Date,FlightNum String,Div5WheelsOff
   String,Div5TailNum String)ENGINE = MergeTree() PARTITION BY toYYYYMM(FlightDate) PRIMARY
   KEY (intHash32(FlightDate)) ORDER BY (intHash32(FlightDate),FlightNum) SAMPLE BY
   intHash32(FlightDate) SETTINGS index_granularity= 8192;
   ```
2. Add the **test** column to table **DB_table1**.
   ```
   ALTER TABLE DB_table1 ADD COLUMN test String DEFAULT 'defaultvalue';
   ```
   Query the table.
   ```
   desc DB_tables;
   ```
3. Change the type of the **Year** column in the **DB_table1** table to UInt8.
   ```
   ALTER TABLE DB_table1 MODIFY COLUMN Year UInt8;
   ```
   View the table structure.
   ```
   desc DB_tables;
   ```
4. Delete the **test** column from the **DB_table1** table.
   ```
   ALTER TABLE DB_table1 DROP COLUMN test;
   ```
   Query the table.
   ```
   desc DB_tables;
   ```
5. Change the name of the **Month** column in the **DB_table1** table to **Month_test**.
   ```
   ALTER TABLE DB_table1 RENAME COLUMN Month to Month_test;
   ```
   Query the table.
   ```
   desc DB_tables;
   ```

2.2.10 DROP

This section describes the basic syntax and usage of the SQL statement for deleting a ClickHouse table.

Basic Syntax

```
DROP [TEMPORARY] TABLE [IF EXISTS] [database_name.]name [ON CLUSTER cluster] [SYNC]
```

Examples:

- Delete the **t1** table.
  ```
  drop table t1 SYNC;
  ```

  NOTE

  - When you delete a replication table, create a path on ZooKeeper to store related data. The default library engine of ClickHouse is the atomic database engine. After a table in the atomic database is deleted, it is not deleted immediately but deleted 24 hours later. To resolve this issue, when deleting a table, add the **SYNC** field to the deletion command, for example, **drop table** *t1* **SYNC;**.
  - This issue does not occur when a local or distributed table is deleted. The **SYNC** field is not required in your deletion command, for example, **drop table** *t1***;**.
  - If the table creation statement contains **ON CLUSTER** *ClickHouse cluster name*, run the following command to delete the table:
    ```
    drop table Table name ON CLUSTER default_cluster;
    ```
  - If the table creation statement does not contain **ON CLUSTER** *ClickHouse cluster name*, run the following command to delete the table:
    ```
    drop table Table name;
    ```
  - Before deleting a data table, check whether the data table is in use to avoid unnecessary troubles. After a data table is deleted, it can be restored within 24 hours. The restoration command is as follows:
    ```
    set allow_experimental_undrop_table_query = 1;
    UNDROP TABLE Data table name;
    ```

2.2.11 SHOW

This section describes the basic syntax and usage of the SQL statement for displaying information about databases and tables in ClickHouse.

Basic Syntax

```
show databases;
show tables;
```

Examples:

- Query the database.
  ```
  show databases;
  ```
- Query the table information.
  ```
  show tables;
  ```

2.3 Data Migration and Synchronization

2.3.1 Importing and Exporting data

This section describes the basic syntax and usage of the SQL statements for importing and exporting file data using ClickHouse.

Importing and Exporting Data in CSV Format

- Import data in CSV format.
  - For normal clusters:
    ```
    cat csv_ssl | ./clickhouse client --host 192.168.x.x --port port --user admin --password password
    --database test010 --query="INSERT INTO test145 FORMAT CSV"
    ```
  - For security clusters:
    ```
    cat csv_no_ssl | ./clickhouse client --host 192.168.x.x --port port --user admin --password
    password --config-file ./config.xml --database test010 --query="INSERT INTO test146 FORMAT
    CSV"
    ```

1. **host**: indicates the host name or ClickHouse instance IP address.
2. **port**: indicates the port number (available on the cluster details page).
3. **user**: indicates the username created during cluster creation.
4. **database**: indicates the database name.
5. **password**: indicates the password created during cluster creation.
6. **INSERT INTO**: Enter the target data table behind this parameter.
7. **cat** *File path*: indicates the file storage path, which can be customized.
8. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.

- Export data in CSV format.
  - For normal clusters:
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password Password --database
    test010 -m --query="select * from test139 FORMAT CSV" > ./csv_no_ssl
    ```
  - For security clusters:
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-
    file ./config.xml --database test010 -m --query="select * from test139 FORMAT CSV" > ./
    csv_no_ssl
    ```

1. **host**: indicates the host name or ClickHouse instance IP address.
2. **port**: indicates the port number (available on the cluster details page).
3. **user**: indicates the username created during cluster creation.
4. **database**: indicates the database name.
5. **password**: indicates the password created during cluster creation.
6. **SELECT * FROM**: Enter the target data table behind this parameter.
7. **./csv_no_ssl**: indicates the file storage path, which can be customized.
8. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.

Importing and Exporting Data in Parquet Format

- Import data in Parquet format.
  - For normal clusters:
    ```
    cat parquet_no_ssl.parquet | ./clickhouse client --host 192.168.x.x --port port --user admin --
    password password --database test010 --query="INSERT INTO test145 FORMAT Parquet"
    ```

- For security clusters:
```
cat parquet_no_ssl.parquet | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 --query="INSERT INTO test146 FORMAT Parquet"
```

1. **parquet_no_ssl.parquet**: indicates the path for storing the files, which can be customized.
2. **host**: indicates the host name or ClickHouse instance IP address.
3. **port**: indicates the port number (available on the cluster details page).
4. **user**: indicates the username created during cluster creation.
5. **database**: indicates the database name.
6. **password**: indicates the password created during cluster creation.
7. **INSERT INTO**: Enter the target data table behind this parameter.
8. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.

- Export data in Parquet format.
  - For normal clusters:
```
./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 -m --query="select * from test139 FORMAT Parquet" > ./parquet_no_ssl.parquet
```
  - For security clusters:
```
./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 -m --query="select * from test139 FORMAT Parquet" > ./parquet_ssl.parque
```

1. **host**: indicates the host name or ClickHouse instance IP address.
2. **port**: indicates the port number (available on the cluster details page).
3. **user**: indicates the username created during cluster creation.
4. **database**: indicates the database name.
5. **password**: indicates the password created during cluster creation.
6. **select * from**: Enter the target data table behind this parameter.
7. **./parquet_no_ssl.parquet**: indicates the path for storing the exported Parquet files, which can be customized.
8. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.

Importing and Exporting Data in ORC Format

- Import data in ORC format.
  - For normal clusters:
```
cat orc_no_ssl.orc | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 --query="INSERT INTO test143 FORMAT ORC"
```
  - For security clusters:
```
cat orc_no_ssl.orc | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 --query="INSERT INTO test144 FORMAT ORC
```

1. **cat orc_no_ssl.orc**: path for storing the ORC files, which can be customized.
2. **host**: indicates the host name or ClickHouse instance IP address.
3. **port**: indicates the port number (available on the cluster details page).
4. **user**: indicates the username created during cluster creation.

5. **database**: indicates the database name.
6. **password**: indicates the password created during cluster creation.
7. **INSERT INTO**: Enter the target data table behind this parameter.
8. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.

- Export data in ORC format.
  - For security clusters:
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 -m --query="select * from test139 FORMAT ORC" > ./orc_ssl.orc
    ```
  - For normal clusters:
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 -m --query="select * from test139 FORMAT ORC" > ./orc_no_ssl.orc
    ```

1. **host**: indicates the host name or ClickHouse instance IP address.
2. **port**: indicates the port number (available on the cluster details page).
3. **user**: indicates the username created during cluster creation.
4. **database**: indicates the database name.
5. **password**: indicates the password created during cluster creation.
6. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.
7. **select * from**: Enter the target data table behind this parameter.
8. **/opt/student.orc**: path for storing the exported ORC file, which can be customized.

Importing and Exporting Data in JSON Format

- Import data in JSON format.
  - For normal clusters:
    ```
    cat ./jsonnossl.json | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 --query="INSERT INTO test141 FORMAT JSON"
    ```
  - For security clusters:
    ```
    cat ./jsonssl.json | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 --query="INSERT INTO test142 FORMAT JSON"
    ```

1. **cat** *File path*: indicates the source path, which can be customized.
2. **host**: indicates the host name or ClickHouse instance IP address.
3. **port**: indicates the port number (available on the cluster details page).
4. **user**: indicates the username created during cluster creation.
5. **database**: indicates the database name.
6. **password**: indicates the password created during cluster creation.
7. **INSERT INTO**: Enter the target data table behind this parameter.
8. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.

- Export data in JSON format.
  - For security clusters:
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 -m --query="select * from test139 FORMAT JSON" > ./jsonnossl.json
    ```

– For normal clusters:
```
./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 -m --query="select * from test139 FORMAT JSON" > ./jsonssl.json
```

1. **host**: indicates the host name or ClickHouse instance IP address.
2. **port**: indicates the port number (available on the cluster details page).
3. **user**: indicates the username created during cluster creation.
4. **database**: indicates the database name.
5. **password**: indicates the password created during cluster creation.
6. **SELECT * FROM**: Enter the target data table behind this parameter.
7. **./jsonssl.json**: indicates the destination path, which can be customized.
8. **config-file ./config.xml**: indicates the configuration file. For details, see **ClickHouse Secure Channel**.

2.3.2 Accessing RDS MySQL Using ClickHouse

ClickHouse provides efficient data analysis in OLAP scenarios. It can map a table on the remote database server to the ClickHouse cluster through a database engine such as MySQL, so data can be analyzed in the ClickHouse cluster. The following describes how to interconnect the ClickHouse cluster with the MySQL database instance of RDS.

Prerequisites

- You have prepared the RDS database instance to be interconnected with and the username and password of the database. For details, see **Step 1: Set Up for RDS**.
- A ClickHouse cluster has been created and is running properly.

Constraints

- The RDS database instance and ClickHouse cluster are in the same VPC and subnet.
- Before synchronizing data, you need to evaluate the impact on the performance of the source and destination databases. You are advised to synchronize data during off-peak hours.
- Currently, ClickHouse can interconnect with MySQL and PostgreSQL instances of RDS, but cannot interconnect with SQL Server instances.

Interconnecting ClickHouse with RDS Using the MySQL Engine

The MySQL engine is used to map tables on the remote MySQL server to ClickHouse and allows you to run INSERT and SELECT statements on tables to facilitate data exchange between ClickHouse and MySQL.

- Syntax for using the MySQL engine:
```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER cluster]
ENGINE = MySQL('host:port', ['database' | database], 'user', 'password')
```

Table 2-14 Parameters of the MySQL database

Parameter	Description
hostport	IP address and port number of the RDS MySQL database instance.
database	RDS MySQL database
user	Username of the RDS MySQL database.
password	Password of the RDS MySQL database user.

Example of using the MySQL engine:

a. Connect to the MySQL database of RDS. For details, see **Connect to a DB Instance**.

b. Create a table in the MySQL database and insert data into the table.

c. Run the client command to connect to ClickHouse.

   Use the following command to connect to a normal cluster.

```
./clickhouse client --host Private IP address of the cluster   --port Port --user admin --
password Password
```

   Use the following command to connect to a security cluster. For details, see **ClickHouse Secure Channel**.

```
./clickhouse client --host Private IP address of the cluster --port port --user admin --
password Password --secure --config-file /root/config.xml
```

   NOTE

   Private IP Address: cluster access address on the cluster details page. Replace it with the access address of the cluster you purchased.

d. Create a MySQL database in ClickHouse. After the database is created, it automatically exchanges data with a MySQL server.

```
CREATE DATABASE mysql_db ENGINE = MySQL('IP address of the RDS MySQL database
instance:Port number of the MySQL database instance', 'MySQL database name', 'MySQL
database username', 'Password of the MySQL database user');
```

e. Switch to the created database **mysql_db**.

```
USE mysql_db;
```

   Query the table data in the MySQL database in ClickHouse.

```
SELECT * FROM mysql_table;
┌─int_id─┬─float─┐
│      1 │     2 │
└────────┴───────┘
```

   Data can be properly queried after being inserted.

```
INSERT INTO mysql_table VALUES (3,4);
SELECT * FROM mysql_table;
┌─int_id─┬─float─┐
│      1 │     2 │
│      3 │     4 │
└────────┴───────┘
```

2.4 Development Process

2.4.1 Typical Use Case

This section describes the application development in a typical scenario, helping you quickly learn and master the ClickHouse development process and know key functions.

Scenario

Assume that a user needs to develop an application to store or query the name, age, and onboarding date of a person based on specified search criteria. The procedure is as follows:

1. Set up the database connection.
2. Create an information table.
3. Insert data. (Data in the sample code is randomly generated.)
4. Query the data based on specified search criteria.

2.4.2 Development Guidelines

As an independent DBMS system, ClickHouse allows you to use the SQL language to perform common operations. In the development program example, the clickhouse-jdbc API is used for description.

- **Setting Properties**: Set the parameters for connecting to a ClickHouse service instance.
- **Setting Up a Connection**: Set a connection to the ClickHouse service instance.
- **Creating a Database**: Create a ClickHouse database.
- **Creating a Table**: Create a table in the ClickHouse database.
- **Inserting Data**: Insert data into the ClickHouse table.
- **Querying Data**: Query data in the ClickHouse table.
- **Deleting a Table**: Delete a ClickHouse table.

2.4.3 Preparing Development and Operating Environment

Preparing a Development Environment

Table 1 lists the development and running environment to be prepared for application development.

Table 2-15 Development environment

Item	Description
OS	● Development environment: Windows 7 or later. ● Running environment: Linux If the program needs to be commissioned locally, the running environment must be able to communicate with network on the cluster service plane.

Item	Description
JDK	Install JDK 1.8.0_272.
IntelliJ IDEA installation and configuration	Basic configuration of the development environment. The version must be 2019.1 or other compatible versions. **NOTE** ● If you are using an IBM JDK, ensure that the JDK configured in IntelliJ IDEA is the IBM JDK. ● If you are using an Oracle JDK, ensure that the JDK configured in IntelliJ IDEA is the Oracle JDK. ● If you are using an open JDK, ensure that the JDK configured in IntelliJ IDEA is the Open JDK. ● Do not use the same workspace and the sample project in the same path for different IntelliJ IDEA projects.
Maven installation	Basic configuration of the development environment. It can be used for project management throughout the lifecycle of software development.
Development user	Prepare the ClickHouse cluster user for application development and grant permissions to the user.
7-zip	Tool used to decompress ***.zip** and ***.rar** files. **7-Zip 16.04** is supported.

2.4.4 Configuring and Importing a Sample Project

Background Information

Obtain the ClickHouse development sample project and import the project to IntelliJ IDEA to learn the sample project.

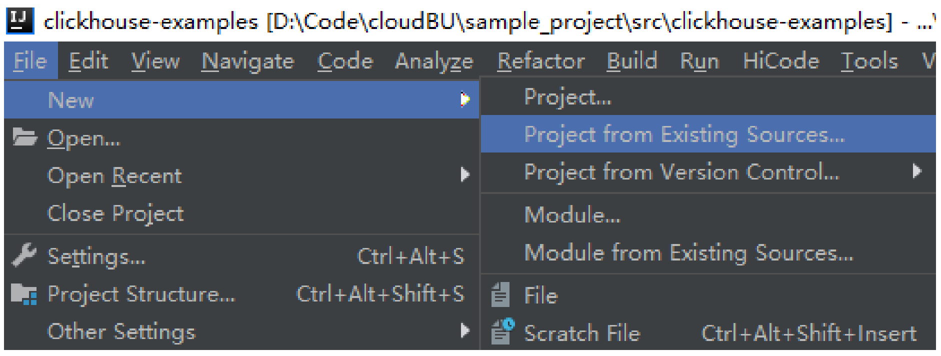
Scenario

ClickHouse provides sample projects for multiple scenarios to help you quickly learn ClickHouse projects.

Procedure

Step 1 In the application development environment, import the sample project to the IntelliJ IDEA development environment.

1. On the IDEA page, choose **File** > **New** > **Project from Existing Sources**.


clickhouse-examples [D:\Code\cloudBU\sample_project\src\clickhouse-examples] - ...
File
Edit
View
Navigate
Code
Analyze
Refactor
Build
Run
HiCode
Tools
New
Open...
Open Recent
Close Project
Settings... Ctrl+Alt+S
Project Structure... Ctrl+Alt+Shift+S
Other Settings
Project...
Project from Existing Sources...
Project from Version Control...
Module...
Module from Existing Sources...
File
Scratch File Ctrl+Alt+Shift+Insert

2. In the displayed **Select File or Directory to Import** dialog box, select the **pom.xml** file in the **clickhouse-examples** folder and click **OK**.
3. Confirm subsequent configurations and click **Next**. If there is no special requirement, use the default values.
4. Select the recommended JDK version and click **Finish**.

Step 2 After the project is imported, modify the **clickhouse-example.properties** file in the **conf** directory of the sample project based on the actual environment information.

```
ipList=
sslUsed=false
httpPort=8123
httpsPort=
CLICKHOUSE_SECURITY_ENABLED=false
user=default
password=
clusterName=default_cluster
databaseName=testdb
tableName=testtb
batchRows=10000
batchNum=10
clickhouse_dataSource_ip_list=ip:8123,ip:8123
native_dataSource_ip_list=ip:9000,ip:9000
```

Table 2-16 Configuration parameters

Parameter	Default Value	Description
iPList	-	Cluster access address list of the clickhouse node. This parameter is mandatory. Log in to the CloudTable console, click the cluster name to go to the cluster details page, and obtain the cluster access address. Use commas (,) to separate multiple addresses, for example, **cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com,cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com**.
sslUsed	false	Whether to enable SSL encryption. The default value is **false**.
httpPort	8123	HTTP port number for connection. The value is **8123**.
httpsPort	-	HTTPS port used for connection. The value is 8443.
CLICKHOUSE_SECURITY_ENABLED	false	Whether to enable the security mode. Set this parameter to **false** for a cluster in normal mode.
user	default	Development user.
password	-	Password of the development user.

Parameter	Default Value	Description
clusterName	default_cluster	ClickHouse logical cluster name. Retain the default value.
databaseName	testdb	Name of the database to be created in the sample code project. You can change the database name based on the site requirements.
tableName	testtb	Name of the table to be created in the sample code project. You can change the table name based on the site requirements.
batchRows	10000	Number of data records written in a batch.
batchNum	10	Total number of batches in which data is written.
clickhouse_dataSource_ip_list	-	List of IP addresses and HTTP ports of the ClickHouse node, for example, **cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123,cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com:8123**.
native_dataSource_ip_list	-	List of IP addresses and TCP ports of the ClickHouse node, for example, **cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:9000,cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com:9000**.

----End

2.4.5 Sample Code

2.4.5.1 Setting Properties

Function Description

Set the connection properties. In the following sample code, the socket timeout interval is set to 60s.

Sample Code

```
Properties clickHouseProperties = new Properties();
clickHouseProperties.setProperty(ClickHouseClientOption.CONNECTION_TIMEOUT.getKey(),
Integer.toString(60000));
clickHouseProperties.setProperty(ClickHouseClientOption.SSL.getKey(), Boolean.toString(false));
clickHouseProperties.setProperty(ClickHouseClientOption.SSL_MODE.getKey(), "none");
```

2.4.5.2 Setting Up a Connection

Function Description

During connection creation, the user and password configured in **clickhouse-example.properties** are used as authentication credentials. ClickHouse performs security authentication on the server with the user and password.

Sample Code

```
ClickHouseDataSource clickHouseDataSource =new ClickHouseDataSource(JDBC_PREFIX +
serverList.get(tries - 1), clickHouseProperties);
connection = clickHouseDataSource.getConnection(user, password);
```

NOTE

There will be huge security risks if the passwords used for authentication are directly written into the code. You are advised to store the password in ciphertext in the configuration file or environment variables and decrypt them when using them.

2.4.5.3 Creating a Database

Function Description

Run the **on cluster** statement to create a database whose name is the value of **databaseName** of **clickhouse-example.properties** in the cluster.

Sample Code

```
private void createDatabase(String databaseName, String clusterName) throws Exception  {
        String createDbSql = "create database if not exists " + databaseName + " on cluster " + clusterName;
        util.exeSql(createDbSql);
}
```

2.4.5.4 Creating a Table

Function Description

Run the **on cluster** statement to create the ReplicatedMerge and Distributed tables whose names are the same as the values of **tableName** in **clickhouse-example.properties**.

Sample Code

```
private void createTable(String databaseName, String tableName, String clusterName) throws Exception {
    String createSql = "create table " + databaseName + "." + tableName + " on cluster " + clusterName
            + " (name String, age UInt8, date Date)engine=ReplicatedMergeTree('/clickhouse/tables/{shard}/" +
databaseName
            + "." + tableName + "'," + "'{replica}') partition by toYYYYMM(date) order by age";
    String createDisSql = "create table " + databaseName + "." + tableName + "_all" + " on cluster " +
clusterName + " as "
            + databaseName + "." + tableName + " ENGINE = Distributed(default_cluster," + databaseName +
"," + tableName + ", rand());";
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(createSql);
    sqlList.add(createDisSql);
    util.exeSql(sqlList);
}
```

2.4.5.5 Inserting Data

Function Description

The created table has three fields of the String, UInt8, and Date types.

Sample Code

```
String insertSql = "insert into " + databaseName + "." + tableName + " values (?,?,?)";
PreparedStatement preparedStatement = connection.prepareStatement(insertSql);
long allBatchBegin = System.currentTimeMillis();
for (int j = 0; j < batchNum; j++) {
    for (int i = 0; i < batchRows; i++) {
        preparedStatement.setString(1, "huawei_" + (i + j * 10));
        preparedStatement.setInt(2, ((int) (Math.random() * 100)));
        preparedStatement.setDate(3, generateRandomDate("2018-01-01", "2021-12-31"));
        preparedStatement.addBatch();
    }
    long begin = System.currentTimeMillis();
    preparedStatement.executeBatch();
    long end = System.currentTimeMillis();
    log.info("Inert batch time is {} ms", end - begin);
}
long allBatchEnd = System.currentTimeMillis();
log.info("Inert all batch time is {} ms", allBatchEnd - allBatchBegin);
```

2.4.5.6 Querying Data

Function Description

Query statement 1 **querySql1** queries any 10 records in the **tableName** table created by the new table.

Query statement 2 **querySql2** uses a built-in function to combine the year and month fields in the **tableName** table.

Sample Code

```
private void queryData(String databaseName, String tableName) throws Exception {
    String querySql1 = "select * from " + databaseName + "." + tableName + "_all" + " order by age limit
10";
    String querySql2 = "select toYYYYMM(date),count(1) from " + databaseName + "." + tableName + "_all"
+ " group by toYYYYMM(date) order by count(1) DESC limit 10";
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(querySql1);
    sqlList.add(querySql2);
    ArrayList<ArrayList<ArrayList<String>>> result = util.exeSql(sqlList);
    for (ArrayList<ArrayList<String>> singleResult : result) {
        for (ArrayList<String> strings : singleResult) {
            StringBuilder stringBuilder = new StringBuilder();
            for (String string : strings) {
                stringBuilder.append(string).append("\t");
            }
            log.info(stringBuilder.toString());
        }
    }
 }
```

2.4.5.7 Deleting a Table

Function Description

Delete the replica table and distributed table created in the new table.

Sample Code

```
private void dropTable(String databaseName, String tableName, String clusterName) throws Exception {
    String dropLocalTableSql = "drop table if exists " + databaseName + "." + tableName + " on cluster " +
clusterName;
    String dropDisTableSql = "drop table if exists " + databaseName + "." + tableName + "_all" + " on
cluster " + clusterName;
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(dropLocalTableSql);
    sqlList.add(dropDisTableSql);
    util.exeSql(sqlList);
 }
```

2.5 Commissioning Applications

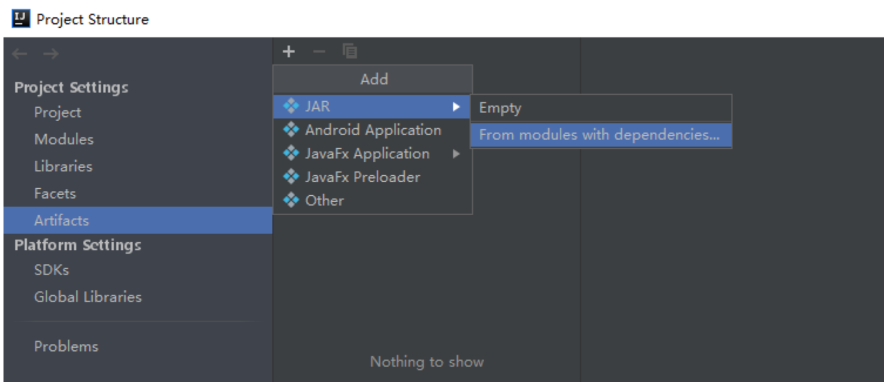
ClickHouse applications can run in a Linux environment. After the application code is developed, you can upload the JAR package to the prepared Linux environment. The environment must be in the same VPC and security group as the clickhouse cluster to ensure network connectivity.

Prerequisites

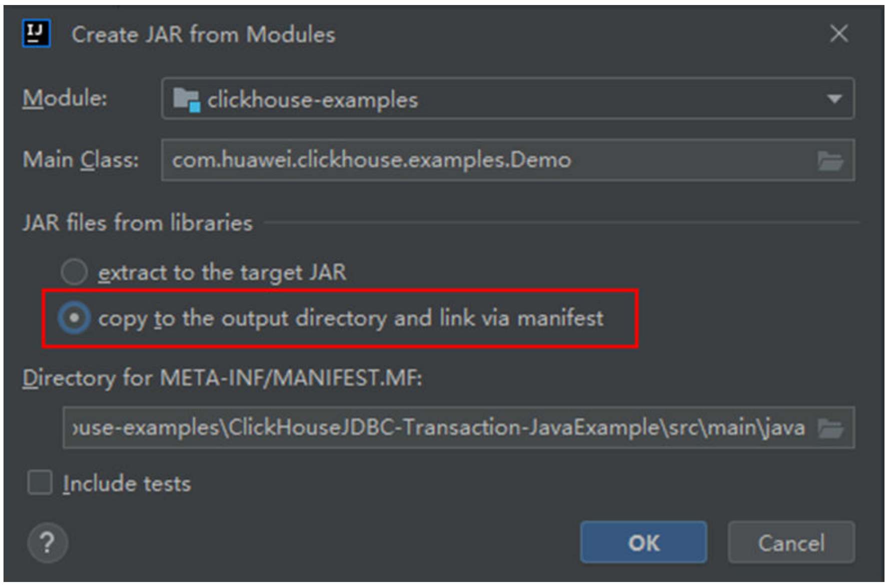
JDK has been installed on Linux. The version must be the same as JDK version of the JAR file exported from IntelliJ IDEA. Java environment variables have been set.

Compiling and Running Applications

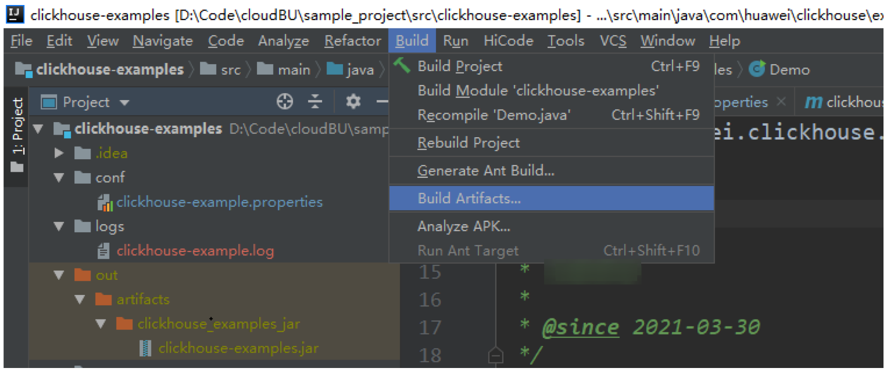
1. Export the JAR file.
   a. Log in to IntelliJ IDEA and choose **File** > **Project Structure** > **Artifacts**.
   b. Click the plus sign (+) and choose **JAR** > **From modules with dependencies**.



   c. Choose **com.huawei.clickhouse.examples.Demo** from the **Main Class** drop-down list and click **OK**.


Create JAR from Modules
Module: clickhouse-examples
Main Class: com.huawei.clickhouse.examples.Demo
JAR files from libraries
extract to the target JAR
copy to the output directory and link via manifest
Directory for META-INF/MANIFEST.MF:
›use-examples\ClickHouseJDBC-Transaction-JavaExample\src\main\java
Include tests
OK
Cancel

d. Choose **Build** > **Build Artifacts...**. After the compilation is successful, view and obtain all JAR files in the **clickhouse-examples\out\artifacts\clickhouse_examples_jar** directory.



2. Copy all JAR files in the **clickhouse-examples\out\artifacts\clickhouse_examples.jar** directory and the **conf** folder in the **clickhouse-examples** directory to the same directory of the ECS.
3. Log in to the client node, go to the directory where the JAR file is uploaded, and change the file permission to 700.

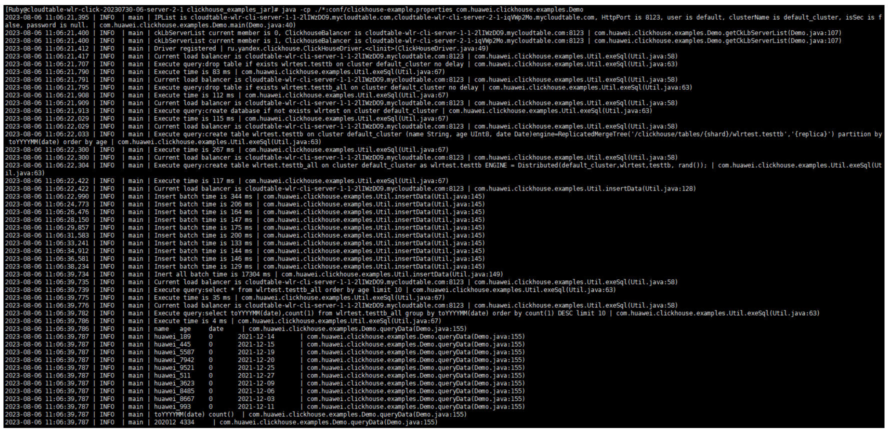
   ```
   chmod -R 700 clickhouse-examples.jar
   ```
4. In the client directory where **clickhouse_examples.jar** is stored, run the following commands to run the JAR file:

   ```
   java -cp ./*:conf/clickhouse-example.properties
   com.huawei.clickhouse.examples.Demo
   ```

Viewing Commissioning Results

If no exception or failure information is displayed, the application running is successful.

Figure 2-3 Run log



```
[Ruby@cloudtable-wlr-click-20230730-06-server-2-1 clickhouse_examples_jar]# java -cp ./*:conf/clickhouse-example.properties com.huawei.clickhouse.examples.Demo
2023-08-06 11:06:21,395 | INFO  | main | IPList is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com,cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com, HttpPort is 8123, user is default, clusterName is default_cluster, isSec is f
alse, password is null. | com.huawei.clickhouse.examples.Demo.main(Demo.java:40)
2023-08-06 11:06:21,400 | INFO  | main | ckLbServerList current member is 0, ClickhouseBalancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Demo.getCkLbServerList(Demo.java:107)
2023-08-06 11:06:21,400 | INFO  | main | ckLbServerList current member is 1, ClickhouseBalancer is cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Demo.getCkLbServerList(Demo.java:107)
2023-08-06 11:06:21,412 | INFO  | main | Driver registered | ru.yandex.clickhouse.ClickHouseDriver.<clinit>(ClickHouseDriver.java:49)
2023-08-06 11:06:21,417 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2023-08-06 11:06:21,707 | INFO  | main | Execute query:drop table if exists wlrtest.testtb on cluster default_cluster no delay | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2023-08-06 11:06:21,790 | INFO  | main | Execute time is 83 ms | com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2023-08-06 11:06:21,791 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2023-08-06 11:06:21,795 | INFO  | main | Execute query:drop table if exists wlrtest.testtb_all on cluster default_cluster no delay | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2023-08-06 11:06:21,908 | INFO  | main | Execute time is 112 ms | com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2023-08-06 11:06:21,909 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2023-08-06 11:06:21,913 | INFO  | main | Execute query:create database if not exists wlrtest on cluster default_cluster | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2023-08-06 11:06:22,029 | INFO  | main | Execute time is 115 ms | com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2023-08-06 11:06:22,029 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2023-08-06 11:06:22,033 | INFO  | main | Execute query:create table wlrtest.testtb on cluster default_cluster (name String, age UInt8, date Date)engine=ReplicatedMergeTree('/clickhouse/tables/{shard}/wlrtest.testtb','{replica}') partition by
 toYYYYMM(date) order by age | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2023-08-06 11:06:22,300 | INFO  | main | Execute time is 267 ms | com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2023-08-06 11:06:22,300 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2023-08-06 11:06:22,304 | INFO  | main | Execute query:create table wlrtest.testtb_all on cluster default_cluster as wlrtest.testtb ENGINE = Distributed(default_cluster,wlrtest,testtb, rand()); | com.huawei.clickhouse.examples.Util.exeSql(Ut
il.java:63)
2023-08-06 11:06:22,422 | INFO  | main | Execute time is 117 ms | com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2023-08-06 11:06:22,422 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.insertData(Util.java:128)
2023-08-06 11:06:22,990 | INFO  | main | Insert batch time is 344 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:24,773 | INFO  | main | Insert batch time is 206 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:26,476 | INFO  | main | Insert batch time is 164 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:28,150 | INFO  | main | Insert batch time is 147 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:29,857 | INFO  | main | Insert batch time is 175 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:31,583 | INFO  | main | Insert batch time is 200 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:33,241 | INFO  | main | Insert batch time is 133 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:34,912 | INFO  | main | Insert batch time is 144 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:36,581 | INFO  | main | Insert batch time is 146 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:38,234 | INFO  | main | Insert batch time is 129 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2023-08-06 11:06:39,734 | INFO  | main | Inert all batch time is 17304 ms | com.huawei.clickhouse.examples.Util.insertData(Util.java:149)
2023-08-06 11:06:39,735 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2023-08-06 11:06:39,739 | INFO  | main | Execute query:select * from wlrtest.testtb_all order by age limit 10 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2023-08-06 11:06:39,775 | INFO  | main | Execute time is 35 ms | com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2023-08-06 11:06:39,776 | INFO  | main | Current load balancer is cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2023-08-06 11:06:39,782 | INFO  | main | Execute query:select toYYYYMM(date),count(1) from wlrtest.testtb_all group by toYYYYMM(date) order by count(1) DESC limit 10 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2023-08-06 11:06:39,786 | INFO  | main | Execute time is 4 ms | com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2023-08-06 11:06:39,786 | INFO  | main | name   age     date     | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_189     0       2021-12-14       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_445     0       2021-12-15       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_5587    0       2021-12-19       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_7942    0       2021-12-20       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_9521    0       2021-12-25       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_511     0       2021-12-27       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_3623    0       2021-12-09       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_8485    0       2021-12-06       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_8667    0       2021-12-03       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | huawei_993     0       2021-12-11       | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | toYYYYMM(date) count()  | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
2023-08-06 11:06:39,787 | INFO  | main | 202012 4334     | com.huawei.clickhouse.examples.Demo.queryData(Demo.java:155)
```

2.6 Developing ClickHouse Cold-Hot Separation Applications

2.6.1 Application Background

CloudTable ClickHouse supports hot and cold data separation. With this feature, you can store hot and cold data in different types of storage media to reduce storage costs.

- Hot Data: This type of data experiences frequent access and updates. It is likely to be needed in future operations and demands swift response times due to its active nature.
- Cold Data: In contrast, cold data is characterized by its static state; it is rarely updated or accessed and has minimal requirements for response speed.

2.6.2 Typical Application Scenarios

You can quickly learn and master the development process of ClickHouse cold and hot data separation and know the functions of key APIs in a typical use case.

Description

Suppose a user creates a web system and uses the table **test_tbl** to log website visits in real-time. The following table lists an example of the recorded data.

Table 2-17 Raw data

timestamp	type	error_code	error_msg	op_id	op_time
2024-06-04 10:36:00	1	404	Resource Not Found	998756	2024-06-04 11:36:00

timestamp	type	error_code	error_msg	op_id	op_time
2024-06-04 10:35:00	1	404	Resource Not Found	998756	2024-06-04 11:35:00
2024-06-03 10:33:00	1	404	Resource Not Found	998756	2024-06-03 11:33:00
2024-03-27 09:10:00	1	200	ok	998756	2024-03-27 10:10:00
2024-03-25 11:08:00	1	404	Resource Not Found	998756	2024-03-25 12:08:00

Data Planning

Data is written on the hour of the current day. The data of the previous day is seldom accessed and is automatically archived to the cold storage to save storage space.

2.6.3 Development Guidelines

Function Description

The functions required are sorted based on the services in the **Typical Scenarios**, as shown below:

Table 2-18 Cold and hot data separation

Procedure	Code Implementation
Step 1: Create a ClickHouse cold and hot separation table.	For details, see **Create a ClickHouse cold and hot data separation table**.
Step 2: Insert data.	For details, see **Insert data for verification**.
Step 3: Query the inserted data.	For details, see **Query inserted data**.

2.6.4 Example Code

This section describes the commands for separating cold data from hot data in CloudTable ClickHouse and the automated transfer of cold data into OBS storage buckets..

Sample Code

- Create the ClickHouse cold and hot data separation table **test_table**.

```
CREATE TABLE IF NOT EXISTS test_table
(

   `timestamp` DATETIME NOT NULL COMMENT " Log time",
    `type` INT NOT NULL COMMENT " log type",
    `error_code` INT COMMENT "Error code",
    `error_msg` VARCHAR(1024) COMMENT "Error details",
    `op_id` BIGINT COMMENT "Operator ID",
    `op_time` DATETIME COMMENT "Operation time"

)
ENGINE = MergeTree()
PARTITION BY timestamp
ORDER BY timestamp

TTL timestamp + INTERVAL 1 DAY TO DISK 'cold_disk'
SETTINGS storage_policy = 'hot_to_cold';
```

- Insert data for verification.

```
insert into test_table values('2024-06-04 10:36:00','1','404','Resource Not Found','998756','2024-06-04 11:36:00');  -- hot data
insert into test_table values('2024-06-04 10:35:00','1','404','Resource Not Found','998756','2024-06-04 11:35:00');  -- hot data
insert into test_table values('2024-06-03 10:33:00','1','404','Resource Not Found','998756','2024-06-03 11:33:00');  -- cold data
insert into test_table values('2024-03-27 09:10:00','1','200','ok','998756','2024-03-27 10:10:00');  -- cold data
insert into test_table values('2024-03-25 11:08:00','1','404','Resource Not Found','998756','2024-03-25 12:08:00');  -- cold data
```

- Query the inserted data.

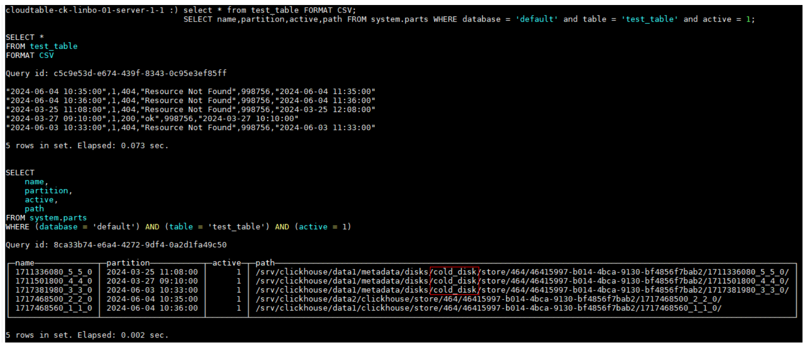
  Query data.

```
select * from test_table FORMAT CSV;
```

  Query the partition fields, partition name, and storage path of the partitioned table used for data storage.

```
SELECT name,partition,active,path FROM system.parts WHERE database = 'default' and table = 'test_table' and active = 1;
```

Figure 2-4 Querying data



```
cloudtable-ck-linbo-01-server-1-1 :) select * from test_table FORMAT CSV;
                                     SELECT name,partition,active,path FROM system.parts WHERE database = 'default' and table = 'test_table' and active = 1;

SELECT *
FROM test_table
FORMAT CSV

Query id: c5c9e53d-e674-439f-8343-0c95e3ef85ff

"2024-06-04 10:35:00",1,404,"Resource Not Found",998756,"2024-06-04 11:35:00"
"2024-06-04 10:36:00",1,404,"Resource Not Found",998756,"2024-06-04 11:36:00"
"2024-03-25 11:08:00",1,404,"Resource Not Found",998756,"2024-03-25 12:08:00"
"2024-03-27 09:10:00",1,200,"ok",998756,"2024-03-27 10:10:00"
"2024-06-03 10:33:00",1,404,"Resource Not Found",998756,"2024-06-03 11:33:00"

5 rows in set. Elapsed: 0.073 sec.


SELECT
    name,
    partition,
    active,
    path
FROM system.parts
WHERE (database = 'default') AND (table = 'test_table') AND (active = 1)

Query id: 8ca33b74-e6a4-4272-9df4-0a2d1fa49c50

┌─name─────────────┬─partition───────────┬─active─┬─path─────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│ 1711336080_5_5_0 │ 2024-03-25 11:08:00 │      1 │ /srv/clickhouse/data1/metadata/disks/cold_disk/store/464/46415997-b014-4bca-9130-bf4856f7bab2/1711336080_5_5_0/ │
│ 1711501800_4_4_0 │ 2024-03-27 09:10:00 │      1 │ /srv/clickhouse/data1/metadata/disks/cold_disk/store/464/46415997-b014-4bca-9130-bf4856f7bab2/1711501800_4_4_0/ │
│ 1717381980_3_3_0 │ 2024-06-03 10:33:00 │      1 │ /srv/clickhouse/data1/metadata/disks/cold_disk/store/464/46415997-b014-4bca-9130-bf4856f7bab2/1717381980_3_3_0/ │
│ 1717468500_2_2_0 │ 2024-06-04 10:35:00 │      1 │ /srv/clickhouse/data2/clickhouse/store/464/46415997-b014-4bca-9130-bf4856f7bab2/1717468500_2_2_0/                │
│ 1717468560_1_1_0 │ 2024-06-04 10:36:00 │      1 │ /srv/clickhouse/data1/clickhouse/store/464/46415997-b014-4bca-9130-bf4856f7bab2/1717468560_1_1_0/                │
└──────────────────┴─────────────────────┴────────┴─────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘

5 rows in set. Elapsed: 0.002 sec.
```

The current system time is 22:00 on June 4, 2024. Data in the **timestamp** column of the **test_table** table that has been stored for more than one day is transferred to the OBS bucket **cold_disk** for storage.

A Change History

Released On	Description
2024-06-20	This issue is the second official issue. Added the following sections: ● **Developing ClickHouse Cold-Hot Separation Applications**
2024-04-30	This issue is the first official release.