Data Lake Insight

Developer Guide

Issue 01

Date 2025-02-21

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

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 Connecting to DLI Using a Client..... 1
1.1 Submitting a SQL Job Using JDBC..... 1
1.1.1 Downloading and Installing the JDBC Driver Package..... 1
1.1.2 Connecting to DLI and Submitting SQL Jobs Using JDBC..... 4
1.1.3 APIs Supported By the DLI JDBC Driver..... 9
1.2 Using Spark-submit to Submit a Spark Jar Job..... 11
1.3 Submitting a Spark Jar Job Using Livy..... 15
2 SQL Jobs..... 21
2.1 Using Spark SQL Jobs to Analyze OBS Data..... 21
2.2 Developing a DLI SQL Job in DataArts Studio..... 32
2.3 Calling UDFs in Spark SQL Jobs..... 44
2.4 Calling UDTFs in Spark SQL Jobs..... 52
2.5 Calling UDAFs in Spark SQL Jobs..... 60
3 Flink Jobs..... 69
3.1 Stream Ecosystem..... 69
3.2 Flink OpenSource SQL Jobs..... 69
3.2.1 Reading Data from Kafka and Writing Data to RDS..... 69
3.2.2 Reading Data from Kafka and Writing Data to GaussDB(DWS)..... 76
3.2.3 Reading Data from Kafka and Writing Data to Elasticsearch..... 82
3.2.4 Reading Data from MySQL CDC and Writing Data to GaussDB(DWS)..... 89
3.2.5 Reading Data from PostgreSQL CDC and Writing Data to GaussDB(DWS)..... 95
3.2.6 Configuring High-Reliability Flink Jobs (Automatic Restart upon Exceptions)..... 104
3.3 Flink Jar Job Examples..... 107
3.4 Writing Data to OBS Using Flink Jar..... 115
3.5 Using Flink Jar to Connect to Kafka that Uses SASL_SSL Authentication..... 123
3.6 Using Flink Jar to Read and Write Data from and to DIS..... 134
3.7 Flink Job Agencies..... 144
3.7.1 Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials..... 144
3.7.2 Flink Jar Jobs Using DEW to Acquire Access Credentials for Reading and Writing Data from and to OBS..... 147
3.7.3 Obtaining Temporary Credentials from a Flink Job's Agency for Accessing Other Cloud Services..... 152
4 Spark Jar Jobs..... 156

4.1 Using Spark Jar Jobs to Read and Query OBS Data....156
4.2 Using the Spark Job to Access DLI Metadata....169
4.3 Using Spark Jobs to Access Data Sources of Datasource Connections....186
4.3.1 Overview....186
4.3.2 Connecting to CSS....186
4.3.2.1 CSS Security Cluster Configuration....186
4.3.2.2 Scala Example Code....188
4.3.2.3 PySpark Example Code....199
4.3.2.4 Java Example Code....206
4.3.3 Connecting to GaussDB(DWS)....212
4.3.3.1 Scala Example Code....212
4.3.3.2 PySpark Example Code....221
4.3.3.3 Java Example Code....225
4.3.4 Connecting to HBase....226
4.3.4.1 MRS Configuration....226
4.3.4.2 Scala Example Code....227
4.3.4.3 PySpark Example Code....233
4.3.4.4 Java Example Code....238
4.3.4.5 Troubleshooting....242
4.3.5 Connecting to OpenTSDB....243
4.3.5.1 Scala Example Code....243
4.3.5.2 PySpark Example Code....246
4.3.5.3 Java Example Code....249
4.3.5.4 Troubleshooting....251
4.3.6 Connecting to RDS....251
4.3.6.1 Scala Example Code....251
4.3.6.2 PySpark Example Code....261
4.3.6.3 Java Example Code....264
4.3.7 Connecting to Redis....266
4.3.7.1 Scala Example Code....266
4.3.7.2 PySpark Example Code....273
4.3.7.3 Java Example Code....276
4.3.7.4 Troubleshooting....278
4.3.8 Connecting to Mongo....278
4.3.8.1 Scala Example Code....278
4.3.8.2 PySpark Example Code....283
4.3.8.3 Java Example Code....286
4.4 Spark Jar Jobs Using DEW to Acquire Access Credentials for Reading and Writing Data from and to OBS....288
4.5 Obtaining Temporary Credentials from a Spark Job's Agency for Accessing Other Cloud Services....292

1 Connecting to DLI Using a Client

1.1 Submitting a SQL Job Using JDBC

This section describes how to connect to DLI and submit a SQL job using JDBC.

1.1.1 Downloading and Installing the JDBC Driver Package

Scenario

To connect to DLI, JDBC is utilized. You can obtain the JDBC installation package from Maven or download the JDBC driver file from the DLI management console.

This section describes how to connect to DLI and submit a SQL job using JDBC.

Obtaining the Server Connection Address

The address for connecting to DLI is in the format of **jdbc:dli://***<endPoint>*/*<projectId>*. So, you need to obtain the endpoint and project ID.

Obtain the DLI endpoint from **Regions and Endpoints**. Specifically, log in to the public cloud, hover over your username in the upper right corner, and choose **My Credentials** from the shortcut menu. You can obtain the project ID on the displayed **API Credentials** tab page.

Downloading and Installing the JDBC Driver

NOTE

Once JDBC 2.X has undergone function reconstruction, query results can only be accessed from DLI job buckets. To utilize this feature, certain conditions must be met:

- On the DLI management console, choose **Global Configuration** > **Project** to configure the job bucket.
- Starting May 2024, new users can directly use DLI's function to write query results into buckets without needing to whitelist it.

  For users who started using DLI before May 2024, to use this function, they must submit a service ticket to whitelist it.

- **Method 1: Adding the JDBC driver using the Maven central repository**

The **Maven central repository** is part of the Apache Maven project that provides Java libraries and frameworks.

When the JDBC retrieval method is not specified, the default approach is to add the JDBC driver using the Maven central repository.

Use Maven to add the Maven configuration item on which **huaweicloud-dli-jdbc** depends. (This is the default operation and does not need to be configured separately.)

```
<dependency>
  <groupId>com.huawei.dli</groupId>
  <artifactId>huaweicloud-dli-jdbc</artifactId>
  <version>x.x.x</version>
</dependency>
```

- **Method 2: Obtaining the JDBC driver using Maven to configure the Huawei image source**

  When using Maven to manage project dependencies, you can modify the **settings.xml** file to configure the Huawei image source to obtain the JDBC driver.

  ```
  <mirror>
    <id>huaweicloud</id>
    <mirrorOf>*</mirrorOf>
    <url>https://mirrors.huaweicloud.com/repository/maven/</url>
  </mirror>
  ```

- **Method 3: Downloading the JDBC driver file from the DLI management console**

  a. Log in to the DLI management console.

  b. Click **SDK Download** in the **Common Links** area on the right of the **Overview** page.

  c. On the **DLI SDK DOWNLOAD** page, select a driver and download it.

  Click **huaweicloud-dli-jdbc-*x.x.x*** to download a JDBC driver package.

  NOTE

  The JDBC driver package is named **huaweicloud-dli-jdbc-*<version>*.zip**. It can be used in all versions of all platforms (such as Linux and Windows) and depends on JDK 1.7 or later.

  d. The downloaded JDBC driver package contains **.bat** (Windows) or **.sh** (Linux/Mac) scripts, which are used to automatically install the JDBC driver to the local Maven repository.

  You can choose a script based on your OS to install the JDBC driver.

  - Windows: Double-click the **.bat** file or run the file in the CLI.
  - Linux/Mac: Run the **.sh** script.

Authentication

You need to be authenticated when using JDBC to create DLI driver connections.

JDBC currently supports two authentication modes: AK/SK-based and token-based. Token-based authentication is only supported by **dli-jdbc-1.*x***. AK/SK-based authentication is recommended.

- **(Recommended) Generating an AK/SK**

a. Log in to the DLI management console.

b. Hover over the username in the upper right corner and select **My Credentials** from the shortcut list.

c. The **Projects** area is displayed on the **API Credentials** tab page by default. Choose **Access Keys** in the navigation pane on the left.

d. Click **Create Access Key**. In the dialog box that appears, set **Login Password** and **SMS Verification Code**.

e. Click **OK** to download the certificate.

f. Once the certificate is downloaded, you can obtain the AK and SK information in the **credentials** file.

NOTE

Hard coding AKs and SKs or storing them in code in plaintext poses significant security risks. You are advised to store them in encrypted form in configuration files or environment variables and decrypt them when needed to ensure security.

- **Obtaining a token**

  When using token-based authentication, you need to obtain the user token and configure the token information in the JDBC connection parameters. You can obtain the token as follows:

  a. Send ***POST https://<IAM_Endpoint>/v3/auth/tokens***. To obtain the IAM endpoint and region name in the message body, see **Regions and Endpoints**.

  Here is an example request:

  NOTE

  Replace the content in italic in the sample code with the actual values. For details, see **Identity and Access Management API Reference**.

```
{
  "auth": {
    "identity": {
      "methods": [
        "password"
      ],
      "password": {
        "user": {
          "name": "username",
          "password": "password",
          "domain": {
            "name": "domainname"
          }
        }
      }
    },
    "scope": {
      "project": {
        "id": "0aa253a31a2f4cfda30eaa073fee6477" //Assume that project_id is 0aa253a31a2f4cfda30eaa073fee6477.
      }
    }
  }
}
```

  b. After the request is processed, the value of **X-Subject-Token** in the response header is the token value.

1.1.2 Connecting to DLI and Submitting SQL Jobs Using JDBC

Scenario

In Linux or Windows, you can connect to the DLI server using JDBC.

NOTE

- Jobs submitted to DLI using JDBC are executed on the Spark engine.
- Once JDBC 2.X has undergone function reconstruction, query results can only be accessed from DLI job buckets. To utilize this feature, certain conditions must be met:
  - On the DLI management console, choose **Global Configuration** > **Project** to configure the job bucket.
  - Starting May 2024, new users can directly use DLI's function to write query results into buckets without needing to whitelist it.

    For users who started using DLI before May 2024, to use this function, they must submit a service ticket to whitelist it.

DLI supports 13 data types. Each type can be mapped to a JDBC type. If JDBC is used to connect to the server, you must use the mapped Java type. **Table 1-1** describes the mapping relationships.

Table 1-1 Data type mapping

DLI Data Type	JDBC Type	Java Type
INT	INTEGER	java.lang.Integer
STRING	VARCHAR	java.lang.String
FLOAT	FLOAT	java.lang.Float
DOUBLE	DOUBLE	java.lang.Double
DECIMAL	DECIMAL	java.math.BigDecimal
BOOLEAN	BOOLEAN	java.lang.Boolean
SMALLINT/SHORT	SMALLINT	java.lang.Short
TINYINT	TINYINT	java.lang.Short
BIGINT/LONG	BIGINT	java.lang.Long
TIMESTAMP	TIMESTAMP	java.sql.Timestamp
CHAR	CHAR	Java.lang.Character
VARCHAR	VARCHAR	java.lang.String
DATE	DATE	java.sql.Date

Prerequisites

Before using JDBC, perform the following operations:

1. Getting authorized.

   DLI uses the Identity and Access Management (IAM) to implement fine-grained permissions for your enterprise-level tenants. IAM provides identity authentication, permissions management, and access control, helping you securely access your HUAWEI CLOUD resources.

   With IAM, you can use your HUAWEI CLOUD account to create IAM users for your employees, and assign permissions to the users to control their access to specific resource types.

   Currently, roles (coarse-grained authorization) and policies (fine-grained authorization) are supported. For details about permissions and authorization operations, see the **Data Lake Insight User Guide**.

2. Create a queue. Choose **Resources** > **Queue Management**. On the page displayed, click **Buy Queue** in the upper right corner. On the **Buy Queue** page displayed, select **For general purpose** for **Type**, that is, the compute resources of the Spark job.

   NOTE

   If the user who creates the queue is not an administrator, the queue can be used only after being authorized by the administrator. For details about how to assign permissions, see **Queue Permission Management**.

Procedure

Step 1 Install JDK 1.7 or later on the computer where JDBC is installed, and configure environment variables.

Step 2 Obtain the DLI JDBC driver package **huaweicloud-dli-jdbc-<version>.zip** by referring to **Downloading and Installing the JDBC Driver Package**. Decompress the package to obtain **huaweicloud-dli-jdbc-<version>-jar-with-dependencies.jar**.

Step 3 On the computer using JDBC, add **huaweicloud-dli-jdbc-1.1.1-jar-with-dependencies.jar** to the **classpath** path of the Java project.

Step 4 DLI JDBC provides two authentication modes, namely, token and AK/SK, to connect to DLI. For how to obtain the token and AK/SK, see **Authentication**.

Step 5 Run the **Class.forName()** command to load the DLI JDBC driver.

Class.forName("com.huawei.dli.jdbc.DliDriver");

Step 6 Call the GetConnection method of DriverManager to create a connection.

Connection conn = DriverManager.getConnection(String url, Properties info);

JDBC configuration items are passed using the URL. For details, see **Table 1-2**. JDBC configuration items can be separated by semicolons (;) in the URL, or you can dynamically set the attribute items using the Info object. For details, see **Table 1-3**.

Table 1-2 Database connection parameters

Parameter	Description
url	The URL format is as follows: jdbc:dli://<endPoint>/projectId? <key1>=<val1>;<key2>=<val2>... • **EndPoint** indicates the DLI domain name. **ProjectId** indicates the project ID. To obtain the endpoint corresponding to DLI, see **Regions and Endpoints**. To obtain the project ID, log in to the public cloud, move the mouse on the account, and click **My Credentials** from the shortcut menu. • Other configuration items are listed after **?** in the form of **key=value**. The configuration items are separated by semicolons (;). They can also be passed using the Info object.
Info	The Info object passes user-defined configuration items. If Info does not pass any attribute item, you can set it to null. The format is as follows: info.setProperty ("Attribute item", "Attribute value").

Table 1-3 Attribute items

Item	Mandatory	Default Value	Description	Supported dli-jdbc
queuename	Yes	-	Queue name of DLI.	dli-jdbc-1.x dli-jdbc-2.x
databasename	No	-	Name of a database.	dli-jdbc-1.x dli-jdbc-2.x
authenticationmode	No	token	Authentication mode. Currently, token- and AK/SK-based authentication modes are supported.	dli-jdbc-1.x
accesskey	Yes	-	AK that acts as the authentication key. For how to obtain the AK, see **Authentication**.	dli-jdbc-1.x dli-jdbc-2.x
secretkey	Yes	-	SK that acts as the authentication key. For how to obtain the SK, see **Authentication**.	dli-jdbc-1.x dli-jdbc-2.x

Item	Mandatory	Default Value	Description	Supported dli-jdbc
regionname	This parameter must be configured if **authenticationmode** is set to **aksk**.	-	Region name. For details, see **Regions and Endpoints**.	dli-jdbc-1.x dli-jdbc-2.x
token	This parameter must be configured if **authenticationmode** is set to **token**.	-	Token-based authentication. For details, see **Authentication**.	dli-jdbc-1.x
charset	No	UTF-8	JDBC encoding mode.	dli-jdbc-1.x dli-jdbc-2.x
usehttpproxy	No	false	Whether to use the access proxy.	dli-jdbc-1.x
proxyhost	This parameter must be configured if **usehttpproxy** is set to **true**.	-	Access proxy host.	dli-jdbc-1.x dli-jdbc-2.x
proxyport	This parameter must be configured if **usehttpproxy** is set to **true**.	-	Access proxy port.	dli-jdbc-1.x dli-jdbc-2.x

Item	Mandatory	Default Value	Description	Supported dli-jdbc
dli.sql.checkNoResultQuery	No	false	Whether to allow invoking the executeQuery API to execute statements (for example, DDL) that do not return results. • Value **false** indicates that invoking of the executeQuery API is allowed. • Value **true** indicates that invoking of the executeQuery API is not allowed.	dli-jdbc-1.x dli-jdbc-2.x
jobtimeout	No	300	End time of the job submission. Unit: second	dli-jdbc-1.x dli-jdbc-2.x
directfetchthreshold	No	1000	Check whether the number of returned results exceeds the threshold based on service requirements. The default threshold is **1000**.	dli-jdbc-1.x

Step 7 Create a Statement object, set related parameters, and submit Spark SQL to DLI.

Statement statement = conn.createStatement();

statement.execute("SET
dli.sql.spark.sql.forcePartitionPredicatesOnPartitionedTable.enabled=true");

statement.execute("select * from tb1");

Step 8 Obtain the result.

ResultSet rs = statement.getResultSet();

Step 9 Display the result.

```
while (rs.next()) {
int a = rs.getInt(1);
int b = rs.getInt(2);
}
```

Step 10 Close the connection.

conn.close();

----End

Example

NOTE

- Hard-coded or plaintext AK and SK pose significant security risks. To ensure security, encrypt your AK and SK, store them in configuration files or environment variables, and decrypt them when needed.
- In this example, the AK and SK stored in the environment variables are used. Specify the environment variables **System.getenv("AK")** and **System.getenv("SK")** in the local environment first.

```
import java.sql.*;
import java.util.Properties;

public class DLIJdbcDriverExample {

    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        Connection conn = null;
        try {
            Class.forName("com.huawei.dli.jdbc.DliDriver");
            String url = "jdbc:dli://<endpoint>/<projectId>?databasename=db1;queuename=testqueue";
            Properties info = new Properties();
            info.setProperty("authenticationmode", "aksk");
            info.setProperty("regionname", "<real region name>");
            info.setProperty("accesskey", "<System.getenv("AK")>");
            info.setProperty("secretkey", "<System.getenv("SK")>");
            conn = DriverManager.getConnection(url, info);
            Statement statement = conn.createStatement();
            statement.execute("select * from tb1");
            ResultSet rs = statement.getResultSet();
            int line = 0;
            while (rs.next()) {
                line ++;
                int a = rs.getInt(1);
                int b = rs.getInt(2);
                System.out.println("Line:" + line + ":" + a + "," + b);
            }
            statement.execute("SET dli.sql.spark.sql.forcePartitionPredicatesOnPartitionedTable.enabled=true");
            statement.execute("describe tb1");
            ResultSet rs1 = statement.getResultSet();
            line = 0;
            while (rs1.next()) {
                line ++;
                String a = rs1.getString(1);
                String b = rs1.getString(2);
                System.out.println("Line:" + line + ":" + a + "," + b);
            }
        } catch (SQLException ex) {
        } finally {
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

1.1.3 APIs Supported By the DLI JDBC Driver

The DLI JDBC driver supports multiple APIs of the JDBC standard, but some APIs cannot be invoked by users. For example, when transaction-related API **prepareCall** is invoked, the **SQLFeatureNotSupportedException** exception is reported. For details about the APIs, see the JDBC official website **https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html**.

Supported APIs

The following tables list the APIs supported by the DLI JDBC driver and provide remarks on possible incompatibilities with the JDBC standard.

- Common signatures supported by Connection APIs
  - Statement createStatement()
  - PreparedStatement prepareStatement(String sql)
  - void close()
  - boolean isClosed()
  - DatabaseMetaData getMetaData()
  - PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)
- Common signatures supported by Driver APIs
  - Connection connect(String url, Properties info)
  - boolean acceptsURL(String url)
  - DriverPropertyInfo[] getPropertyInfo(String url, Properties info)
- Common signatures supported by Connection APIs
  - String getColumnClassName(int column)
  - int getColumnCount()
  - int getColumnDisplaySize(int column)
  - String getColumnLabel(int column)
  - String getColumnName(int column)
  - int getColumnType(int column)
  - String getColumnTypeName(int column)
  - int getPrecision(int column)
  - int getScale(int column)
  - boolean isCaseSensitive(int column)
- Common signatures supported by Statement APIs
  - ResultSet executeQuery(String sql)
  - int executeUpdate(String sql)
  - boolean execute(String sql)
  - void close()
  - int getMaxRows()
  - void setMaxRows(int max)
  - int getQueryTimeout()
  - void setQueryTimeout(int seconds)
  - void cancel()
  - ResultSet getResultSet()
  - int getUpdateCount()
  - boolean isClosed()
- Common signatures supported by PreparedStatement APIs

  - void clearParameters()
  - boolean execute()
  - ResultSet executeQuery()
  - int executeUpdate()
  - PreparedStatement Set methods
- Common signatures supported by ResultSet APIs
  - int getRow()
  - boolean isClosed()
  - boolean next()
  - void close()
  - int findColumn(String columnLabel)
  - boolean wasNull()
  - Get methods
- Common signatures supported by DatabaseMetaData APIs
  - ResultSet getCatalogs()

    NOTE

    DLI does not have the concept of Catalog, so an empty ResultSet is returned.
  - ResultSet getColumns(String catalog, String schemaPattern,
    String tableNamePattern, String columnNamePattern)
  - Connection getConnection()
  - getTables(String catalog, String schemaPattern,String tableNamePattern, String types[])

    NOTE

    This method does not use the **Catalog** parameter, and schemaPattern corresponds to the database concept of DLI.
  - ResultSet getTableTypes()
  - ResultSet getSchemas()
  - ResultSet getSchemas(String catalog, String schemaPattern)

1.2 Using Spark-submit to Submit a Spark Jar Job

Introduction to DLI Spark-submit

DLI Spark-submit is a command line tool used to submit Spark jobs to the DLI server. This tool provides command lines compatible with open-source Spark.

Preparations

1. Getting authorized.

   DLI uses the Identity and Access Management (IAM) to implement fine-grained permissions for your enterprise-level tenants. IAM provides identity authentication, permissions management, and access control, helping you securely access your HUAWEI CLOUD resources.

With IAM, you can use your HUAWEI CLOUD account to create IAM users for your employees, and assign permissions to the users to control their access to specific resource types.

Currently, roles (coarse-grained authorization) and policies (fine-grained authorization) are supported. For details about permissions and authorization operations, see the **Data Lake Insight User Guide**.

2. Create a queue. Choose **Resources** > **Queue Management**. On the page displayed, click **Buy Queue** in the upper right corner. On the **Buy Queue** page displayed, select **For general purpose** for **Type**, that is, the compute resources of the Spark job.

   NOTE

   If the user who creates the queue is not an administrator, the queue can be used only after being authorized by the administrator. For details about how to assign permissions, see **Queue Permission Management**.

Downloading the DLI Client Tool

You can download the DLI client tool from the DLI management console.

Step 1 Log in to the DLI management console.

Step 2 Click **SDK Download** in the **Common Links** area on the right of the **Overview** page.

Step 3 On the **DLI SDK DOWNLOAD** page, click **dli-clientkit-<*version*>** to download the DLI client tool.

NOTE

The Beeline client is named **dli-clientkit-<*version*>-bin.tar.gz**, which can be used in Linux and depends on JDK 1.8 or later.

----End

Configuring DLI Spark-submit

Ensure that you have installed JDK 1.8 or later and configured environment variables on the computer where spark-submit is installed. You are advised to use spark-submit on the computer running Linux.

Step 1 Download and decompress **dli-clientkit-<*version*>-bin.tar.gz**. In this step, set *version* to the actual version.

Step 2 Go to the directory where **dli-clientkit-<version>-bin.tar.gz** is decompressed. In the directory, there are three subdirectories **bin**, **conf**, and **lib**, which respectively store the execution scripts, configuration files, and dependency packages related to **Spark-submit**.

Step 3 Go to the **conf** directory and modify the configuration items in the **client.properties** file. For details about the configuration items, see **Table 1-4**.

Table 1-4 DLI client parameters

Item	Mandatory	Default Value	Description
dliEndPont	No	-	Domain name of DLI Obtain the domain name corresponding to the region of DLI at **Regions and Endpoints**. If you lef this parameter empty, the program determines the domain name based on **region**.
obsEndPoint	Yes	-	OBS service domain name. Obtain the domain name corresponding to the region of OBS at **Regions and Endpoints**.
bucketName	Yes	-	Name of a bucket on OBS. This bucket is used to store JAR files, Python program files, and configuration files used in Spark programs.
obsPath	Yes	dli-spark-submit-resources	Directory for storing JAR files, Python program files, and configuration files on OBS. The directory is in the bucket specified by **Bucket Name**. If the directory does not exist, the program automatically creates it.
localFilePath	Yes	-	The local directory for storing JAR files, Python program files, and configuration files used in Spark programs. The program automatically uploads the files on which Spark depends to the OBS path and loads them to the resource package on the DLI server.
ak	Yes	-	User's Access Key (AK)
sk	Yes	-	User's Secret Key (SK)
projectId	Yes	-	Project ID used by a user to access DLI.
region	Yes	-	Region of interconnected DLI.

Modify the configuration items in the **spark-defaults.conf** file based on the Spark application requirements. The configuration items are compatible with the open-source Spark configuration items. For details, see the open-source Spark configuration item description.

----End

Using Spark-submit to Submit a Spark Job

Step 1 Go to the **bin** directory of the tool file, run the **spark-submit** command, and carry related parameters.

The command format is as follows:

```
spark-submit [options] <app jar | python file> [app arguments]
```

Table 1-5 DLI Spark-submit parameters

Parameter	Value	Description
--class	<CLASS_NAME>	Name of the main class of the submitted Java or Scala application.
--conf	<PROP=VALUE>	Spark program parameters can be configured in the **spark-defaults.conf** file in the **conf** directory. If both the command and the configuration file are configured, the parameter value specified in the command is preferentially used. **NOTE** If there are multiple **conf** files, the format is **--conf key1=value1 --conf key2=value2**.
--jars	<JARS>	Name of the JAR file on which the Spark application depends. Use commas (,) to separate multiple names. The JAR file must be stored in the local path specified by **localFilePath** in the **client.properties** file in advance.
--name	<NAME>	Name of a Spark application.
--queue	<QUEUE_NAME>	Name of the Spark queue on the DLI server. Jobs are submitted to the queue for execution.
--py-files	<PY_FILES>	Name of the Python program file on which the Spark application depends. Use commas (,) to separate multiple file names. The Python program file must be saved in the local path specified by **localFilePath** in the **client.properties** file in advance.

Parameter	Value	Description
-s,--skip-upload-resources	<all | app | deps>	Specifies whether to skip. Upload the JAR file, Python program file, and configuration file to OBS and load them to the resource list on the DLI server. If related resource files have been loaded to the DLI resource list, skip this step. If this parameter is not specified, all resource files in the command are uploaded and loaded to DLI by default. • **all**: Skips the upload and loading all resource files. • **app**: Skips the upload and loading of Spark application files. • **deps**: skips the upload and loading of all dependent files.
-h,--help	-	Displays command help information.

Command example:

```
./spark-submit --name <name> --queue <queue_name> --class org.apache.spark.examples.SparkPi spark-examples_2.11-2.1.0.luxor.jar 10
./spark-submit --name <name> --queue <queue_name> word_count.py
```

NOTE

To use the DLI queue rather than the existing Spark environment, use **./spark-submit** instead of **spark-submit**.

----End

1.3 Submitting a Spark Jar Job Using Livy

Introduction to DLI Livy

DLI Livy is an Apache Livy-based client tool used to submit Spark jobs to DLI.

Preparations

- Create a queue. Set **Queue Usage** to **For general purpose**, that is, the computing resources of the Spark job. For details, see **Creating a Queue**.
- Prepare a Linux ECS for installing DLI Livy.
  - Enable ports 30000 to 32767 and port 8998 on the ECS. For details, see **Adding a Security Group Rule**.
  - Install JDK on the ECS. JDK 1.8 is recommended. Configure Java environment variable **JAVA_HOME**.
  - View the ECS details to obtain its private IP address.

- Use an enhanced datasource connection to connect the DLI queue to the VPC where the Livy instance is located. For details, see **Enhanced Datasource Connection**.

Downloading and Installing DLI Livy

NOTE

The software package used in the following operations is **apache-livy-0.7.2.0107-bin.tar.gz**. Replace it with the latest one.

Step 1 **Download** the DLI Livy software package.

Step 2 Use WinSCP to upload the obtained software package to the prepared ECS directory.

Step 3 Log in to ECS as user **root** and perform the following steps to install DLI Livy:

1. Run the following command to create an installation directory:

   mkdir *Livy installation directory*

   For example, to create the **/opt/livy** directory, run the **mkdir /opt/livy** command. The following operations use the **/opt/livy** installation directory as an example. Replace it as required.

2. Run the following command to decompress the software package to the installation directory:

   tar --extract --file apache-livy-0.7.2.0107-bin.tar.gz --directory /opt/livy --strip-components 1 --no-same-owner

3. Run the following commands to change the configuration file name:

   cd /opt/livy/conf

   mv livy-client.conf.template livy-client.conf

   mv livy.conf.template livy.conf

   mv livy-env.sh.template livy-env.sh

   mv log4j.properties.template log4j.properties

   mv spark-blacklist.conf.template spark-blacklist.conf

   touch spark-defaults.conf

----End

Modifying the DLI Livy Configuration File

Step 1 Upload the specified DLI Livy JAR package to the OBS bucket directory.

1. Log in to OBS console and create a directory for storing the DLI Livy JAR package in the specified OBS bucket, for example: **obs://bucket/livy/jars/**.
2. Go to the installation directory of the ECS where the DLI-Livy tool has been installed in **Step 3.1**, obtain Livy JAR packages, and upload them to the OBS bucket directory created in **Step 1.1**:

   For example, if the installation path is **/opt/livy**, the JAR packages you need to upload are as follows:

   ```
   /opt/livy/rsc-jars/livy-api-0.7.2.0107.jar
   /opt/livy/rsc-jars/livy-rsc-0.7.2.0107.jar
   ```



```
/opt/livy/repl_2.11-jars/livy-core_2.11-0.7.2.0107.jar
/opt/livy/repl_2.11-jars/livy-repl_2.11-0.7.2.0107.jar
```

Step 2 Modify the DLI Livy configuration file.

1. Run the following command to modify the **/opt/livy/conf/livy-client.conf** configuration file:

   vi /opt/livy/conf/livy-client.conf

   Add the following content to the file and modify the configuration items as required:

   ```
   # Set the private IP address of the ECS, which can be obtained by running the ifconfig command.
   livy.rsc.launcher.address = X.X.X.X
   # Set the ports enabled on the ECS.
   livy.rsc.launcher.port.range = 30000~32767
   ```

2. Run the following command to modify the **/opt/livy/conf/livy.conf** configuration file:

   vi /opt/livy/conf/livy.conf

   Add the following content to the file and modify the configuration items as required:

   ```
   livy.server.port = 8998
   livy.spark.master = yarn

   livy.server.contextLauncher.custom.class=org.apache.livy.rsc.DliContextLauncher
   livy.server.batch.custom.class=org.apache.livy.server.batch.DliBatchSession
   livy.server.interactive.custom.class=org.apache.livy.server.interactive.DliInteractiveSession
   livy.server.sparkApp.custom.class=org.apache.livy.utils.SparkDliApp

   livy.server.recovery.mode = recovery
   livy.server.recovery.state-store = filesystem
   # Change the following file directory of DLI Livy as needed:
   livy.server.recovery.state-store.url = file:///opt/livy/store/

   livy.server.session.timeout-check = true
   livy.server.session.timeout = 1800s
   livy.server.session.state-retain.sec = 1800s

   livy.dli.spark.version = 2.3.2
   livy.dli.spark.scala-version = 2.11

   # Enter the OBS bucket path that stores the Livy JAR file.
   livy.repl.jars = obs://bucket/livy/jars/livy-core_2.11-0.7.2.0107.jar, obs://bucket/livy/jars/livy-repl_2.11-0.7.2.0107.jar
   livy.rsc.jars = obs://bucket/livy/jars/livy-api-0.7.2.0107.jar, obs://bucket/livy/jars/livy-rsc-0.7.2.0107.jar
   ```

3. Run the following command to modify the **/opt/livy/conf/spark-defaults.conf** configuration file:

   vi /opt/livy/conf/spark-defaults.conf

   Add the following content to the file. Set the parameters based on **Table 1-6**.

   ```
   # The following parameters can be overwritten when a job is submitted.
   spark.yarn.isPython=true
   spark.pyspark.python=python3

   # Enter the production environment URL of DLI.
   spark.dli.user.uiBaseAddress=https://console.huaweicloud.com/dli/web
   # Set the region where the queue is located.
   spark.dli.user.regionName=XXXX

   # Set the DLI endpoint address.
   spark.dli.user.dliEndPoint=XXXX

   # Enter the name of the created DLI queue.
   spark.dli.user.queueName=XXXX
   ```



```
# Set the AK used for submitting a job.
spark.dli.user.access.key=XXXX
# Set the SK used for submitting a job.
spark.dli.user.secret.key=XXXX

# Set the project ID used for submitting a job.
spark.dli.user.projectId=XXXX
```

Table 1-6 Mandatory parameters in spark-defaults.conf

Parameter	Description
spark.dli.user.regionName	Name of the region where the DLI queue is. Obtain the region name from **Regions and Endpoints**.
spark.dli.user.dliEndPoint	Endpoint where the DLI queue is located. Obtain the endpoint from **Regions and Endpoints**.
spark.dli.user.queueName	Queue name.
spark.dli.user.access.key	User's AK/SK. The user must have Spark job permissions. For details, see **Permissions Management**. For how to obtain the AK/SK, see **Obtaining the AK/SK**.
spark.dli.user.secret.key	
spark.dli.user.projectId	Project ID. Obtain it by referring to **Obtaining a Project ID**.

The following parameters are optional. Set them based on the parameter description and site requirements. For details about these parameters, see **Spark Configuration**.

Table 1-7 Optional parameters in spark-defaults.conf

Spark Job Parameter	Spark Batch Processing Parameter	Remarks
spark.dli.user.file	file	Not required for connecting to the notebook tool
spark.dli.user.className	class_name	Not required for connecting to the notebook tool
spark.dli.user.scType	sc_type	Same as the native Livy configuration
spark.dli.user.args	args	Same as the native Livy configuration
spark.submit.pyFiles	python_files	Same as the native Livy configuration

Spark Job Parameter	Spark Batch Processing Parameter	Remarks
spark.files	files	Same as the native Livy configuration
spark.dli.user.modules	modules	-
spark.dli.user.image	image	Custom image used for submitting a job. This parameter is available for container clusters only and is not set by default.
spark.dli.user.autoRecovery	auto_recovery	-
spark.dli.user.maxRetryTimes	max_retry_times	-
spark.dli.user.catalogName	catalog_name	To access metadata, set this parameter to **dli**.

----End

Starting DLI Livy

Step 1 Run the following command to go to the DLI Livy installation directory:

Example: **cd /opt/livy**

Step 2 Run the following command to start DLI Livy:

./bin/livy-server start

----End

Submitting a Spark Job Using DLI Livy

The following demonstrates how to submit a Spark job to DLI using DLI Livy and by running the curl command.

Step 1 Upload the JAR file of the developed Spark job program to the OBS directory.

For example, upload **spark-examples_2.11-XXXX.jar** to the **obs://bucket/path** directory.

Step 2 Log in to the ECS server where DLI Livy is installed as user **root**.

Step 3 Run the **curl** command to submit a Spark job request to DLI using DLI Livy.

NOTE

ECS_IP indicates the private IP address of the ECS where DLI Livy is installed.

```
curl --location --request POST 'http://ECS_IP:8998/batches' \
--header 'Content-Type: application/json' \
```



```
--data '{
    "driverMemory": "3G",
    "driverCores": 1,
    "executorMemory": "2G",
    "executorCores": 1,
    "numExecutors": 1,
    "args": [
        "1000"
    ],
    "file": "obs://bucket/path/spark-examples_2.11-XXXX.jar",
    "className": "org.apache.spark.examples.SparkPi",
    "conf": {
        "spark.dynamicAllocation.minExecutors": 1,
        "spark.executor.instances": 1,
        "spark.dynamicAllocation.initialExecutors": 1,
        "spark.dynamicAllocation.maxExecutors": 2
    }
}'
```

----End

2 SQL Jobs

2.1 Using Spark SQL Jobs to Analyze OBS Data

DLI allows you to use data stored on OBS. You can create OBS tables on DLI to access and process data in your OBS bucket.

This section describes how to create an OBS table on DLI, import data to the table, and insert and query table data.

Prerequisites

- You have created an OBS bucket. For details, see *Object Storage Service User Guide*. In this example, the OBS bucket name is **dli-test-021**.
- You have created a DLI SQL queue. For details, see **Creating a Queue**.

  Note: When you create the DLI queue, set **Type** to **For SQL**.

Preparations

Creating a Database on DLI

1. Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark** and **Queue** to the created SQL queue.
2. Enter the following statement in the SQL editing window to create the **testdb** database. For details about the syntax for creating a DLI database, see **Creating a Database**.

   ```
   create database testdb;
   ```

The following operations in this section must be performed for the **testdb** database.

DataSource and Hive Syntax for Creating an OBS Table on DLI

The main difference between DataSource syntax and Hive syntax lies in the range of table data storage formats supported and the number of partitions supported. For the key differences in creating OBS tables using these two syntax, refer to **Table 2-1**.

Table 2-1 Syntax differences

Syntax	Data Types	Partitioning	Number of Partitions
Data Source	ORC, PARQUET, JSON, CSV, and AVRO	You need to specify the partitioning column in both CREATE TABLE and PARTITIONED BY statements. For details, see **Creating a Single-Partition OBS Table Using DataSource Syntax**.	A maximum of 7,000 partitions can be created in a single table.
Hive	TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, and PARQUET	Do not specify the partitioning column in the CREATE TABLE statement. Specify the column name and data type in the PARTITIONED BY statement. For details, see **Creating an OBS Table Using Hive Syntax**.	A maximum of 100,000 partitions can be created in a single table.

For details about the DataSource syntax, see **Creating an OBS Table Using the DataSource Syntax**.

For details about the Hive syntax, see **Creating an OBS Table Using the Hive Syntax**.

Creating an OBS Table Using the DataSource Syntax

The following describes how to create an OBS table for CSV files. The methods of creating OBS tables for other file formats are similar.

- Create a non-partitioned OBS table.
  - Specify an OBS file and create an OBS table for the CSV data.
    i. Create the **test.csv** file containing the following content and upload the **test.csv** file to the **root** directory of OBS bucket **dli-test-021**:
    ```
    Jordon,88,23
    Kim,87,25
    Henry,76,26
    ```
    ii. Log in to the DLI management console and choose **SQL Editor** from the navigation pane on the left. In the SQL editing window, set **Engine** to **spark**, **Queue** to the SQL queue you have created, and **Database** to **testdb**. Run the following statement to create an OBS table:
    ```
    CREATE TABLE testcsvdatasource (name STRING, score DOUBLE, classNo INT
    ) USING csv OPTIONS (path "obs://dli-test-021/test.csv");
    ```

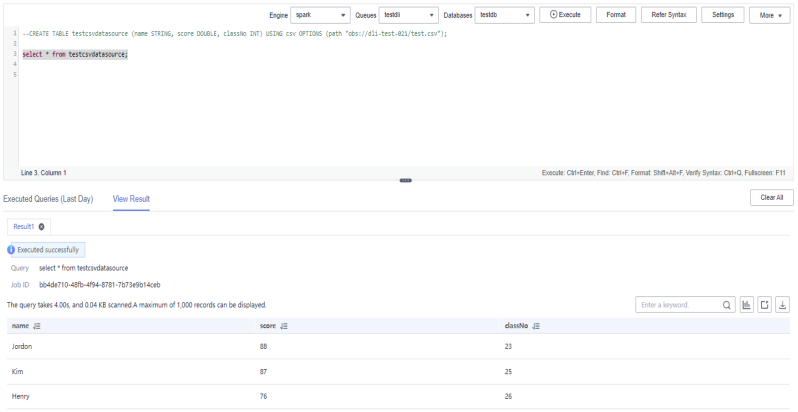

CAUTION

If you create an OBS table using a specified file, you cannot insert data to the table with DLI. The OBS file content is synchronized with the table data.

iii. Run the following statement to query data in the **testcsvdatasource** table.

```
select * from testcsvdatasource;
```

Figure 2-1 Query results



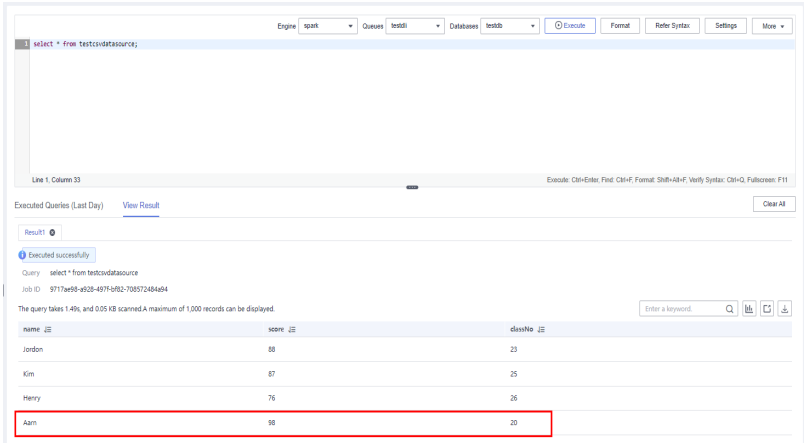
iv. Open the **test.csv** file on the local PC, add **Aarn,98,20** to the file, and replace the original **test.csv** file in the OBS bucket.

```
Jordon,88,23
Kim,87,25
Henry,76,26
Aarn,98,20
```

v. In the DLI **SQL Editor**, query the **testcsvdatasource** table for **Aarn,98,20**. The result is displayed.

```
select * from testcsvdatasource;
```

Figure 2-2 Query results



– Specify an OBS directory and create an OBS table for CSV data.

  - The specified OBS data directory does not contain files you want to import to the table.

1) Create the file directory **data** in the **root** directory of the OBS bucket **dli-test-021**.
2) Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark**, **Queue** to the created SQL queue, and **Database** to **testdb**. Run the following statement to create OBS table **testcsvdata2source** in the **testdb** database on DLI:

```
CREATE TABLE testcsvdata2source (name STRING, score DOUBLE, classNo INT)
USING csv OPTIONS (path "obs://dli-test-021/data");
```

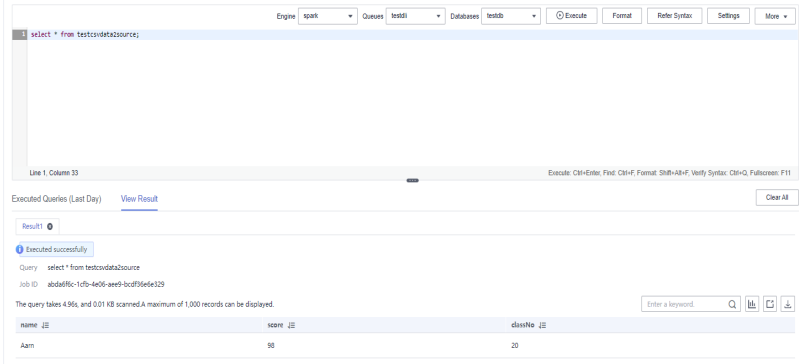
3) Run the following statement to insert table data:

```
insert into testcsvdata2source VALUES('Aarn','98','20');
```

4) Run the following statement to query data in the **testcsvdata2source** table:

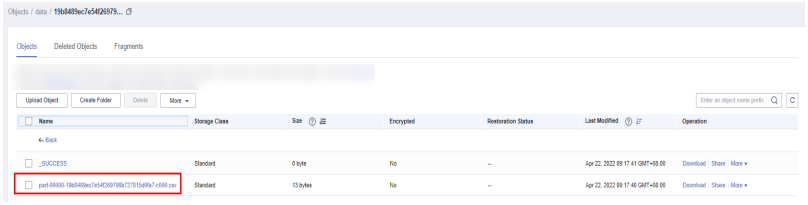
```
select * from testcsvdata2source;
```

Figure 2-3 Query results



5) Refresh the **obs://dli-test-021/data** directory of the OBS bucket and query the data. A CSV data file is generated, and the data is added to the file.

Figure 2-4 Query results



- The specified OBS data directory contains files you want to import to the table.
  1) Create file directory **data2** in the **root** directory of the OBS bucket **dli-test-021**. Create the **test.csv** file with the following content and upload the file to the **obs://dli-test-021/data2** directory:

```
Jordon,88,23
Kim,87,25
Henry,76,26
```

  2) Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark**, **Queue** to the created SQL queue, and **Database** to **testdb**. Run the following statement to create OBS table **testcsvdata3source** in the **testdb** database on DLI:


```
CREATE TABLE testcsvdata3source (name STRING, score DOUBLE, classNo INT)
USING csv OPTIONS (path "obs://dli-test-021/data2");
```

3) Run the following statement to insert table data:

```
insert into testcsvdata3source VALUES('Aarn','98','20');
```

4) Run the following statement to query data in the **testcsvdata3source** table:

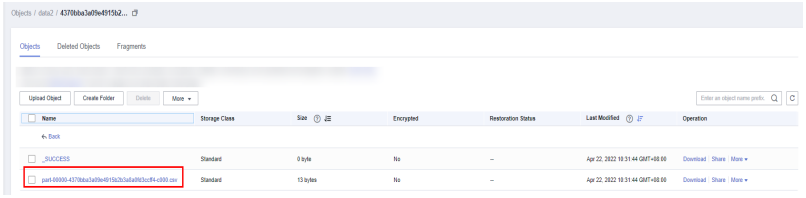
```
select * from testcsvdata3source;
```

Figure 2-5 Query results

name	score	classNo
Jordon	88	23
Kim	87	25
Henry	76	26
Aarn	98	20

5) Refresh the **obs://dli-test-021/data2** directory of the OBS bucket and query the data. A CSV data file is generated, and the data is added to the file.

Figure 2-6 Query results



- Create an OBS partitioned table
  - Create a single-partition OBS table
    i. Create file directory **data3** in the **root** directory of the OBS bucket **dli-test-021**.
    ii. Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark**, **Queue** to the created SQL queue, and **Database** to **testdb**. Run the following statement to create OBS table **testcsvdata4source** using data in the specified OBS directory **obs://dli-test-021/data3** and partition the table on the **classNo** column.

```
CREATE TABLE testcsvdata4source (name STRING, score DOUBLE, classNo INT) USING csv
OPTIONS (path "obs://dli-test-021/data3") PARTITIONED BY (classNo);
```

    iii. Create the **classNo=25** directory in the **obs://dli-test-021/data3** directory of the OBS bucket. Create the **test.csv** file based on the following file content and upload the file to the **obs://dli-test-021/data3/classNo=25** directory of the OBS bucket.

```
Jordon,88,25
Kim,87,25
Henry,76,25
```

    iv. Run the following statement in the SQL editor to add the partition data to OBS table **testcsvdata4source**:

```
ALTER TABLE
  testcsvdata4source
ADD
  PARTITION (classNo = 25) LOCATION 'obs://dli-test-021/data3/classNo=25';
```

    v. Run the following statement to query data in the **classNo=25** partition of the **testcsvdata4source** table:

```
select * from testcsvdata4source where classNo = 25;
```


Figure 2-7 Query results

name	score	classNo
Jordon	88	25
Kim	87	25
Henry	76	25

vi. Run the following statement to insert the following data to the **testcsvdata4source** table:

```
insert into testcsvdata4source VALUES('Aarn','98','25');
insert into testcsvdata4source VALUES('Adam','68','24');
```

vii. Run the following statement to query data in the **classNo=25** and **classNo=24** partitions of the **testcsvdata4source** table:

CAUTION

When a partitioned table is queried using the where condition, the partition must be specified. Otherwise, the query fails and "DLI.0005: There should be at least one partition pruning predicate on partitioned table" is reported.

```
select * from testcsvdata4source where classNo = 25;
```

Figure 2-8 Query results

name	score	classNo
Jordon	88	25
Kim	87	25
Henry	76	25
Aarn	98	25

```
select * from testcsvdata4source where classNo = 24;
```

Figure 2-9 Query results

name	score	classNo
Adam	68	24

viii. In the **obs://dli-test-021/data3** directory of the OBS bucket, click the refresh button. Partition files are generated in the directory for storing the newly inserted table data.

Figure 2-10 classNo=25 file on OBS

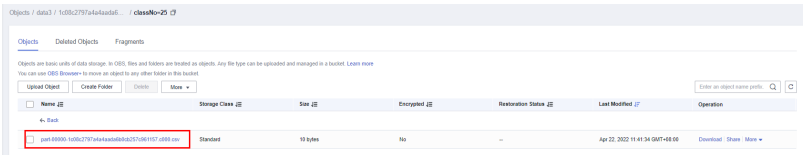
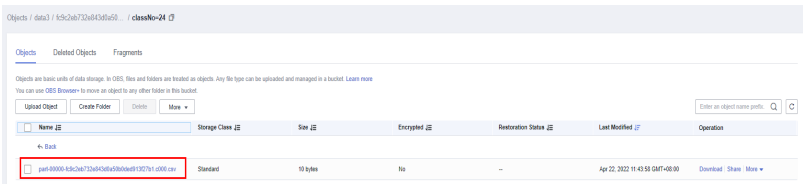


Figure 2-11 classNo=24 file on OBS



- Create an OBS table partitioned on multiple columns.

  i. Create file directory **data4** in the **root** directory of the OBS bucket **dli-test-021**.

  ii. Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark**, **Queue** to the created SQL queue, and **Database** to **testdb**. Run the following statement to create OBS table **testcsvdata5source** using data in the specified OBS directory **obs://dli-test-021/data4** and partition the table on **classNo** and **dt** columns.

  ```
  CREATE TABLE testcsvdata5source (name STRING, score DOUBLE, classNo INT, dt varchar(16)) USING csv OPTIONS (path "obs://dli-test-021/data4") PARTITIONED BY (classNo,dt);
  ```

  iii. Run the following statements to insert the following data into the **testcsvdata5source** table:

  ```
  insert into testcsvdata5source VALUES('Aarn','98','25','2021-07-27');
  insert into testcsvdata5source VALUES('Adam','68','25','2021-07-28');
  ```

  iv. Run the following statement to query data in the **classNo** partition of the **testcsvdata5source** table:

  ```
  select * from testcsvdata5source where classNo = 25;
  ```

  Figure 2-12 Query results

name	score	classNo	dt
Aarn	98	25	2021-07-27
Adam	68	25	2021-07-28

  v. Run the following statement to query data in the **dt** partition of the **testcsvdata5source** table:

  ```
  select * from testcsvdata5source where dt like '2021-07%';
  ```

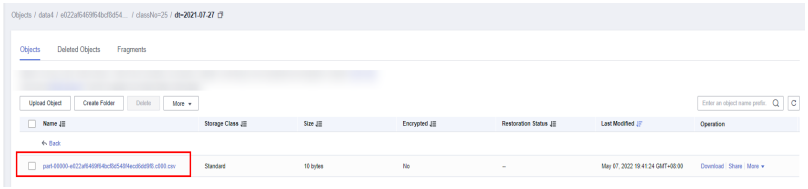
  Figure 2-13 Query results

name	score	classNo	dt
Aarn	98	25	2021-07-27
Adam	68	25	2021-07-28

  vi. Refresh the **obs://dli-test-021/data4** directory of the OBS bucket. The following data files are generated:

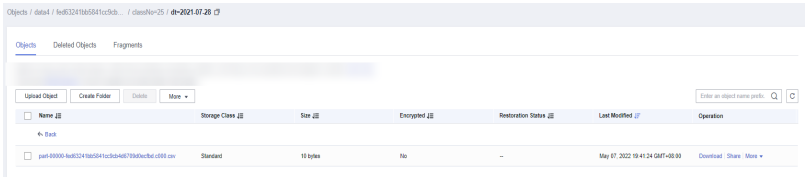
  - File directory 1: **obs://dli-test-021/data4/***xxxxxx***/classNo=25/dt=2021-07-27**

    Figure 2-14 Query results

  - File directory 2: **obs://dli-test-021/data4/***xxxxxx***/classNo=25/dt=2021-07-28**

Figure 2-15 Query results



vii. Create the partition directory **classNo=24** in **obs://dli-test-021/data4**, and then create the subdirectory **dt=2021-07-29** in **classNo=24**. Create the **test.csv** file using the following file content and upload the file to the **obs://dli-test-021/data4/classNo=24/dt=2021-07-29** directory.

```
Jordon,88,24,2021-07-29
Kim,87,24,2021-07-29
Henry,76,24,2021-07-29
```

viii. Run the following statement in the SQL editor to add the partition data to OBS table **testcsvdata5source**:

```
ALTER TABLE
  testcsvdata5source
ADD
  PARTITION (classNo = 24,dt='2021-07-29') LOCATION 'obs://dli-test-021/data4/classNo=24/dt=2021-07-29';
```

ix. Run the following statement to query data in the **classNo** partition of the **testcsvdata5source** table:

```
select * from testcsvdata5source where classNo = 24;
```

Figure 2-16 Query results

name	score	classNo	dt
Jordon	88	24	2021-07-29
Kim	87	24	2021-07-29
Henry	76	24	2021-07-29

x. Run the following statement to query all data in July 2021 in the **dt** partition:

```
select * from testcsvdata5source where dt like '2021-07%';
```

Figure 2-17 Query results

name	score	classNo	dt
Jordon	88	24	2021-07-29
Kim	87	24	2021-07-29
Henry	76	24	2021-07-29
Aarn	98	25	2021-07-27
Adam	68	25	2021-07-28

Creating an OBS Table Using Hive Syntax

The following describes how to create an OBS table for TEXTFILE files. The methods of creating OBS tables for other file formats are similar.

- Create a non-partitioned OBS table.
  a. Create file directory **data5** in the **root** directory of the OBS bucket **dli-test-021**. Create the **test.txt** file based on the following file content and upload the file to the **obs://dli-test-021/data5** directory:

```
Jordon,88,23
Kim,87,25
Henry,76,26
```


b. Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark**, **Queue** to the created SQL queue, and **Database** to **testdb**. Run the following Hive statement to create an OBS table using data in **obs://dli-test-021/data5/test.txt** and set the row data delimiter to commas (,):

```
CREATE TABLE hiveobstable (name STRING, score DOUBLE, classNo INT) STORED AS TEXTFILE
LOCATION 'obs://dli-test-021/data5' ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

> **NOTE**
>
> **ROW FORMAT DELIMITED FIELDS TERMINATED BY ','** indicates that records are separated by commas (,).

c. Run the following statement to query data in the **hiveobstable** table:

```
select * from hiveobstable;
```

Figure 2-18 Query results

name	score	classNo
Jordon	88	23
Kim	87	25
Henry	76	26

d. Run the following statements to insert data into the table:

```
insert into hiveobstable VALUES('Aarn','98','25');
insert into hiveobstable VALUES('Adam','68','25');
```

e. Run the following statement to query data in the table to verify that the data has been inserted:

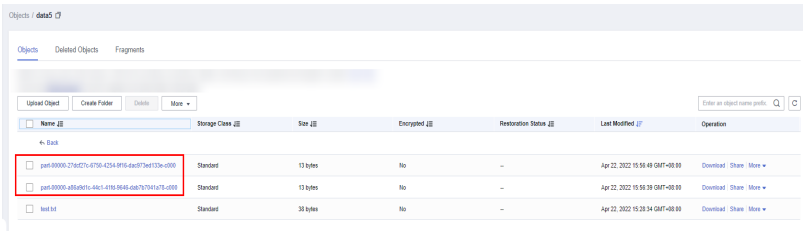
```
select * from hiveobstable;
```

Figure 2-19 Query results

name	score	classNo
Adam	68	25
Aarn	98	25
Jordon	88	23
Kim	87	25
Henry	76	26

f. In the **obs://dli-test-021/data5** directory, refresh the page and query the data. Two files are generated containing the newly inserted data.

Figure 2-20 Query results



Create an OBS Table Containing Data of Multiple Formats

a. Create file directory **data6** in the **root** directory of the OBS bucket **dli-test-021**. Create the **test.txt** file based on the following file content and upload the file to the **obs://dli-test-021/data6** directory:

```
Jordon,88-22,23:21
Kim,87-22,25:22
Henry,76-22,26:23
```


b. Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark**, **Queue** to the created SQL queue, and **Database** to **testdb**. Run the following Hive statement to create an OBS table using data stored in **obs://dli-test-021/data6**.

```
CREATE TABLE hiveobstable2 (name STRING, hobbies ARRAY<string>, address
map<string,string>) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data6'
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '-'
MAP KEYS TERMINATED BY ':';
```

NOTE

- **ROW FORMAT DELIMITED FIELDS TERMINATED BY ','** indicates that records are separated by commas (,).
- **COLLECTION ITEMS TERMINATED BY '-'** indicates that the second column **hobbies** is in array format. Elements are separated by hyphens (-).
- **MAP KEYS TERMINATED BY ':'** indicates that the **address** column is in the key-value format. Key-value pairs are separated by colons (:).

c. Run the following statement to query data in the **hiveobstable2** table:

```
select * from hiveobstable2;
```

Figure 2-21 Query results

name	hobbies	address
Jordon	["88-22"]	{"23":"21"}
Kim	["87-22"]	{"25":"22"}
Henry	["76-22"]	{"26":"23"}

- Create a partitioned OBS table.

a. Create file directory **data7** in the **root** directory of the OBS bucket **dli-test-021**.

b. Log in to the DLI management console and click **SQL Editor**. On the displayed page, set **Engine** to **spark**, **Queue** to the created SQL queue, and **Database** to **testdb**. Run the following statement to create an OBS table using data stored in **obs://dli-test-021/data7** and partition the table on the **classNo** column:

```
CREATE TABLE IF NOT EXISTS hiveobstable3(name STRING, score DOUBLE) PARTITIONED BY
(classNo INT) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7' ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',';
```

CAUTION

You can specify the partition key in the **PARTITIONED BY** statement. Do not specify the partition key in the **CREATE TABLE IF NOT EXISTS** statement. The following is an incorrect example:

CREATE TABLE IF NOT EXISTS hiveobstable3(name STRING, score DOUBLE, classNo INT) PARTITIONED BY (classNo) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7';

c. Run the following statements to insert data into the table:

```
insert into hiveobstable3 VALUES('Aarn','98','25');
insert into hiveobstable3 VALUES('Adam','68','25');
```

d. Run the following statement to query data in the table:

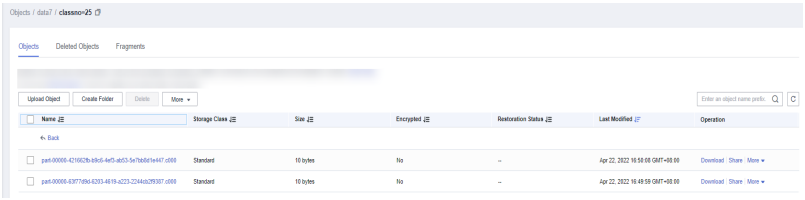
```
select * from hiveobstable3 where classNo = 25;
```


Figure 2-22 Query results

name	score	classNo
Adam	68	25
Aarn	98	25

e. Refresh the **obs://dli-test-021/data7** directory. A new partition directory **classno=25** is generated containing the newly inserted table data.

Figure 2-23 Query results



f. Create partition directory **classno=24** in the **obs://dli-test-021/data7** directory. Create the **test.txt** file using the following file content and upload the file to the **obs://dli-test-021/data7/classno=24** directory:

```
Jordon,88,24
Kim,87,24
Henry,76,24
```

g. Run the following statement in the SQL editor to add the partition data to OBS table **hiveobstable3**:

```
ALTER TABLE
  hiveobstable3
ADD
  PARTITION (classNo = 24) LOCATION 'obs://dli-test-021/data7/classNo=24';
```

h. Run the following statement to query data in the **hiveobstable3** table:

```
select * from hiveobstable3 where classNo = 24;
```

Figure 2-24 Query results

name	score	classNo
Jordon	88	24
Kim	87	24
Henry	76	24

FAQs

- **Q1**: What should I do if the following error is reported when the OBS partition table is queried?

  ```
  DLI.0005: There should be at least one partition pruning predicate on partitioned table `xxxx`.`xxxx`.;
  ```

  Cause: The partition key is not specified in the query statement of a partitioned table.

  Solution: Ensure that the where condition contains at least one partition key.

- **Q2**: What should I do if "DLI.0007: The output path is a file, don't support INSERT...SELECT error" is reported when I use a DataSource statement to insert data in a specified OBS directory into an OBS table and the execution fails?

  The statement is similar to the following:

  ```
  CREATE TABLE testcsvdatasource (name string, id int) USING csv OPTIONS (path "obs://dli-test-021/data/test.csv");
  ```


Cause: Data cannot be inserted if a specific file is used in the table creation statement. For example, the OBS file **obs://dli-test-021/data/test.csv** is used in the preceding example.

Solution: Replace the OBS file to the file directory. You can insert data using the INSERT statement. The preceding example statement can be modified as follows:

```
CREATE TABLE testcsvdatasource (name string, id int) USING csv OPTIONS (path "obs://dli-test-021/data");
```

- **Q3**: What should I do if the syntax of a Hive statement used to create a partitioned OBS table is incorrect? For example, the following statement creates an OBS table partitioned on **classNo**:

  ```
  CREATE TABLE IF NOT EXISTS testtable(name STRING, score DOUBLE, classNo INT) PARTITIONED BY (classNo) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7';
  ```

  Cause: Do not specify the partition key in the list following the table name. Specify the partition key in the **PARTITIONED BY** statement.

  Solution: Specify the partition key in **PARTITIONED BY**. For example:

  ```
  CREATE TABLE IF NOT EXISTS testtable(name STRING, score DOUBLE) PARTITIONED BY (classNo INT) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7';
  ```

2.2 Developing a DLI SQL Job in DataArts Studio

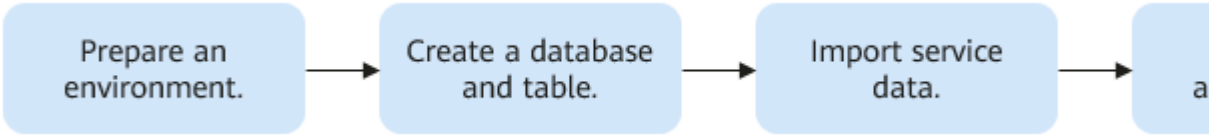
Scenario

Huawei Cloud DataArts Studio provides a one-stop data governance platform that integrates with DLI for seamless data integration and development, enabling enterprises to manage and control their data effectively.

This section walks you through how to develop a DLI SQL job in DataArts Studio.

Development Process

Figure 2-25 Process of developing a DLI SQL job in DataArts Studio



1. Prepare an environment: Prepare DLI and DataArts Studio resources required for job execution. See **Prepare Environments**.
2. Create a database and table: Submit SQL scripts to create a database and table. See **Step 1: Create a Database and Table**.
3. Import service data: Submit SQL scripts to import service data. See **Step 2: Calculate and Process Service Data**.
4. Query and analyze data: Submit SQL scripts to analyze service data, for example, querying daily sales. See **Step 3: Query and Analyze Sales Data**.
5. Orchestrate a job: Orchestrate data processing and analysis scripts into a pipeline. DataArts Studio executes all nodes based on the orchestrated pipeline sequence. See **Step 4: Orchestrate a Job**.
6. Test job runs: Test if jobs can run properly. See **Step 5: Test Job Running**.

7. Configure job scheduling and monitoring: Set job scheduling attributes and monitoring rules. See **Step 6: Set Periodic Job Scheduling** and **Related Operations**.

Prepare Environments

- **Prepare a DLI resource environment.**
  - **Configure a DLI job bucket.**

    Before using DLI, you need to configure a DLI job bucket. The bucket is used to store temporary data generated during DLI job running, such as job logs and results.

    For details, see **Configuring a DLI Job Bucket**.
  - **Create an elastic resource pool and create a SQL queue within it.**

    An elastic resource pool offers compute resources (CPU and memory) required for running DLI jobs, which can adapt to the changing demands of services.

    You can create multiple queues within an elastic resource pool. These queues are associated with specific jobs and data processing tasks, and serve as the basic unit for resource allocation and usage within the pool. This means queues are specific compute resources required for executing jobs.

    Queues within an elastic resource pool can be shared to execute jobs. This is achieved by properly setting the queue allocation policy. This improves queue utilization.

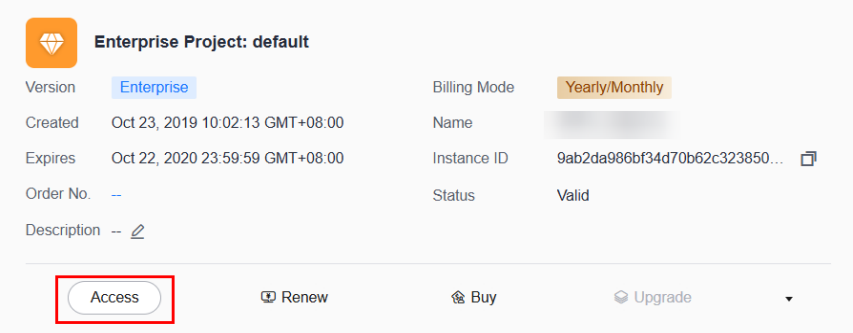
    For details, see **Creating an Elastic Resource Pool and Creating Queues Within It**.
- **Prepare a DataArts Studio resource environment.**
  - **Buy a DataArts Studio instance.**

    Buy a DataArts Studio instance before submitting a DLI job using DataArts Studio.

    For details, see **Buying a DataArts Studio Basic Package**.
  - **Access the DataArts Studio instance's workspace.**

    i. After buying a DataArts Studio instance, click **Access**.

    Figure 2-26 Accessing a DataArts Studio instance

    ii. Click the **Workspaces** tab to access the data development page.

    By default, a workspace named **default** is created for the user who has purchased the DataArts Studio instance, and the user is assigned

the administrator role. You can use the default workspace or create one.

For how to create a workspace, see **Creating and Managing a Workspace**.

Figure 2-27 Accessing the DataArts Studio instance's workspace

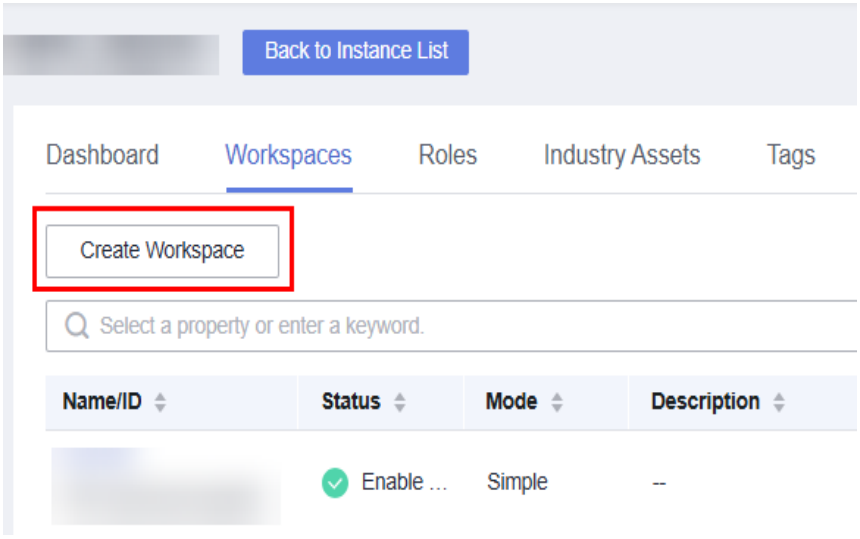
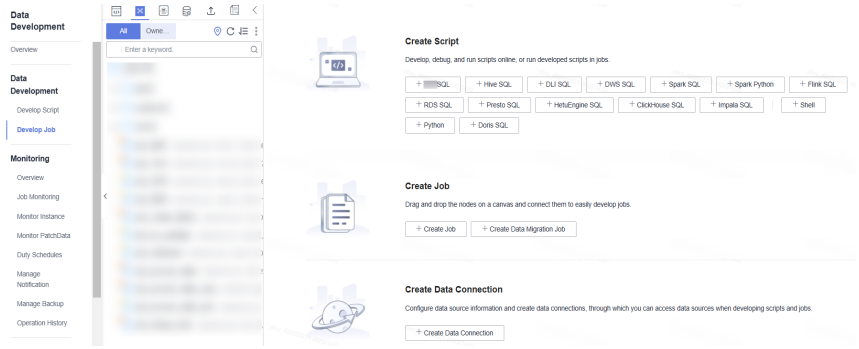


Figure 2-28 Accessing DataArts Studio's data development page



Step 1: Create a Database and Table

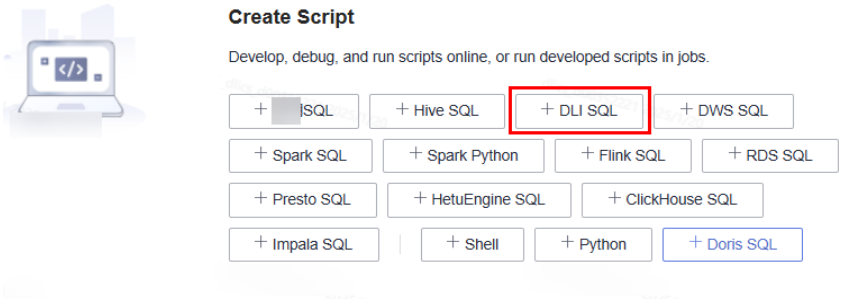
Step 1 Develop SQL scripts for creating databases and tables.

Databases and tables are the basis for developing SQL jobs. Before running a job, you need to define databases and tables based on your service scenarios.

This part describes how to develop a SQL script to create databases and tables.

1. In the left navigation pane of DataArts Factory, choose **Data Development** > **Develop Script**.
2. In the **Create Script** area on the right, click **+ DLI SQL**.

Figure 2-29 Creating a DLI SQL script



3. On the script editing page, enter sample code for creating a database and table.

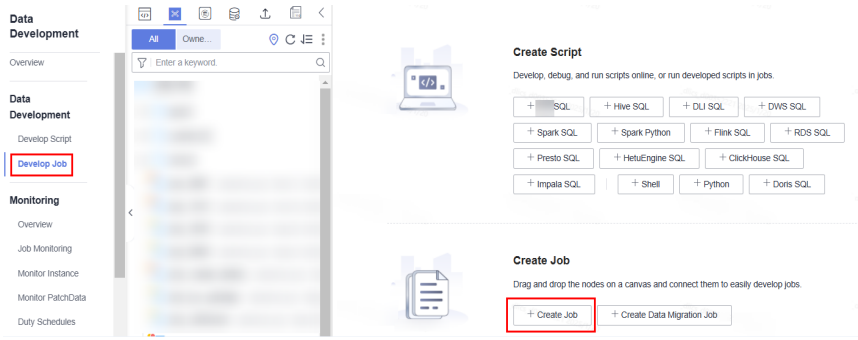
```
```SQL -- Create a database.
CREATE DATABASE IF not EXISTS supermarket_db;-- Create product dimension table.
CREATE TABLE IF not EXISTS supermarketdb.products ( productid INT,  productname STRING, category STRING,  price DECIMAL(10,2) ) using parquet;
-- Create a transaction table. (productid INT, -- Product ID productname STRING, -- Product name category STRING, -- Product category price DECIMAL(10,2) -- Unit price)
CREATE TABLE IF not EXISTS supermarketdb.transactions (transactionid INT,  productid INT,  quantity INT,  dt STRING  ) using parquet partitioned by (dt);
-- Create a sales analysis table. (transactionid INT, -- Transaction ID productid INT, -- Product ID quantity INT, -- Quantity dt STRING -- Date)
CREATE TABLE IF not EXISTS supermarketdb.analyze (transactionid INT, productid INT, productname STRING,  quantity INT, dt STRING ) using parquet partitioned by (dt);
-- (transactionid INT, -- Transaction ID productid INT, -- Product ID productname STRING, -- Product name quantity INT, -- Quantity dt STRING -- Date)
```
```

4. Click **Save** to save the SQL script. In this example, the script is named **create_tables**.
5. Click **Submit** to run the script to create a database and table.

Step 2 Create a SQL job running script.

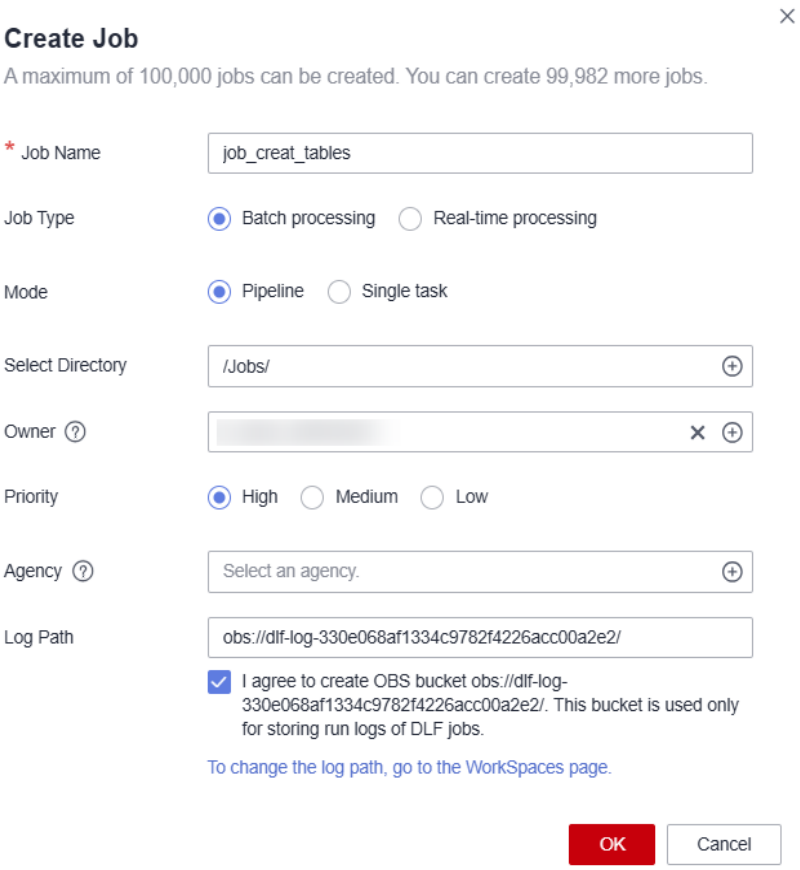
1. In the left navigation pane of DataArts Factory, choose **Data Development** > **Develop Job**.

Figure 2-30 Creating a job



2. Click **Create Job**. In the dialog box that appears, edit job information. In this example, the SQL job is named **job_create_tables**.

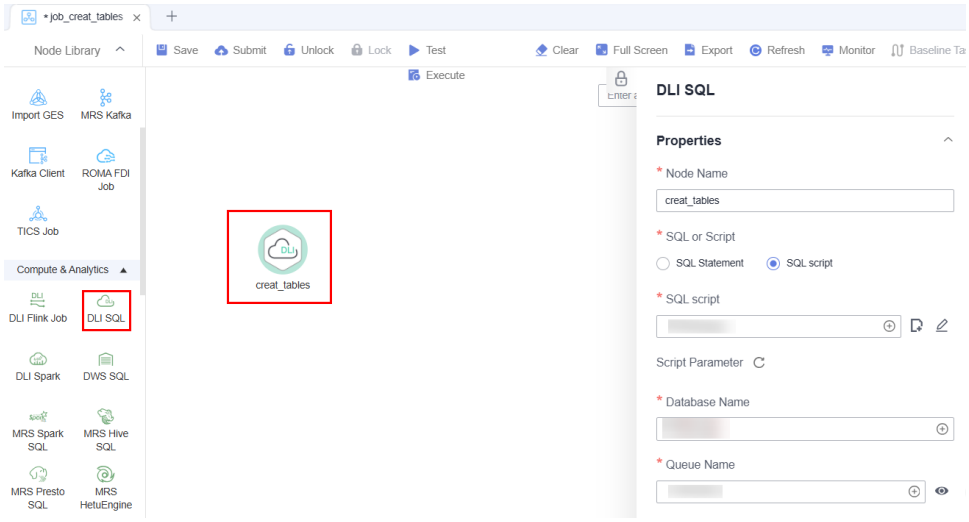
Figure 2-31 Editing job information



3. On the job development page, drag the DLI SQL node to the canvas and click the node to edit its properties.
   - **SQL or Script**: Select **SQL script** in this example. Then, select the script created in **Step 2.2** for **SQL script**.
   - **Database Name**: Select the database configured in the SQL script.
   - **Queue Name**: Select the SQL queue created in **Create an elastic resource pool and create a SQL queue within it**.

   For more property parameters, see **Parameters of DLI SQL nodes**.

Figure 2-32 Editing the properties of a DLI SQL node



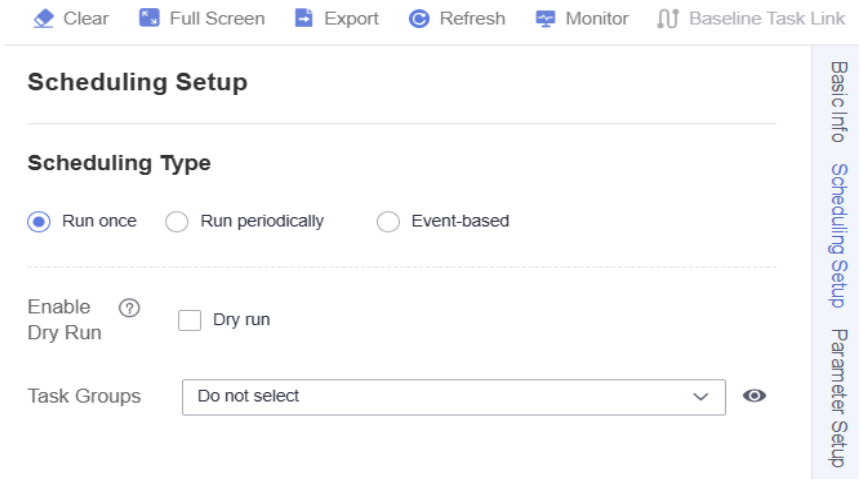
4. Once the properties are edited, click **Save** to save the configuration.

Step 3 Configure job scheduling.

As the database and table only need to be created once, only one-time scheduling is configured in this example.

1. Left-click the blank area of the canvas.
2. Click **Scheduling Setup** and select **Run once**. (The job will be scheduled only once and will not be automatically scheduled later.)

Figure 2-33 Configuring job scheduling



3. After configuring the scheduling, click **Execute**.
   Click **Go to O&M Center** to view the job status.

----End

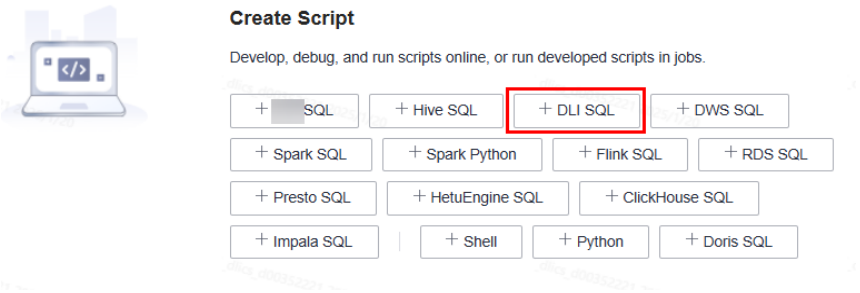
Step 2: Calculate and Process Service Data

Step 1 Develop a SQL script for importing service data.

This part describes how to submit a SQL script to import service data.

1. In the left navigation pane of DataArts Factory, choose **Data Development** > **Develop Script**.
2. In the **Create Script** area on the right, click **+ DLI SQL**.

Figure 2-34 Creating a DLI SQL script



3. On the script editing page, enter sample code for analyzing data.

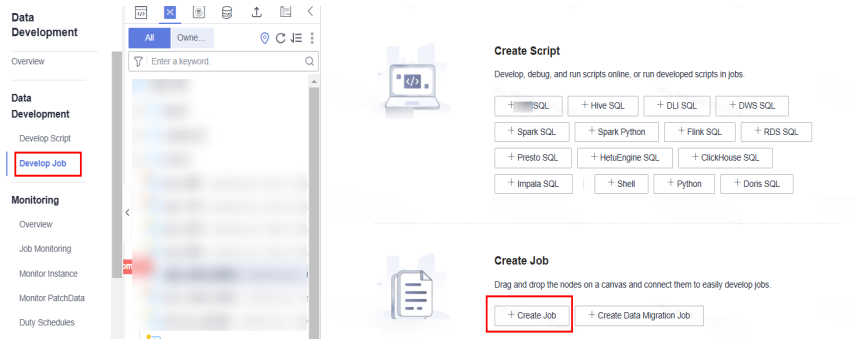
```
SQL: Data in actual services is typically from other data sources. This example simplifies the data import logic and simulates the insertion of product data.
 INSERT INTO supermarketdb.products (productid, productname, category, price) VALUES
(1001, 'Shampoo', 'Daily necessities', 39.90),
(1002, 'Toothpaste', 'Daily necessities', 15.90)
(1003, 'Instant noodles', 'Food', 4.50)
(1004, 'Coke', 'Beverage', 3.50);
-- Data in actual services is typically from other data sources. This example simplifies the data import logic and simulates the insertion of transaction records.
INSERT INTO supermarketdb.transactions (transactionid, productid, quantity, dt) VALUES (1, 1001, 50, '2024-11-01'), -- 50 bottles of shampoo were sold.
(2, 1002, 100, '2024-11-01'), -- 100 tubes of toothpaste were sold.
(3, 1003, 30, '2024-11-02'), -- 30 packs of instant noodles were sold.
(4, 1004, 24, '2024-11-02'); -- 24 bottles of Coke were sold.
-- Simulate supermarket business analysis and query the transaction records of a certain product.
INSERT INTO supermarketdb.analyze SELECT t.transactionid, t.productid, p.productname, t.quantity, t.dt
FROM supermarketdb.transactions t
JOIN supermarketdb.products p ON t.productid = p.productid
WHERE t.dt = '2024-11-01';
```

4. Click **Save** to save the SQL script. In this example, the script is named **job_process_data**.
5. Click **Submit** to execute the script.

Step 2 Create a SQL job.

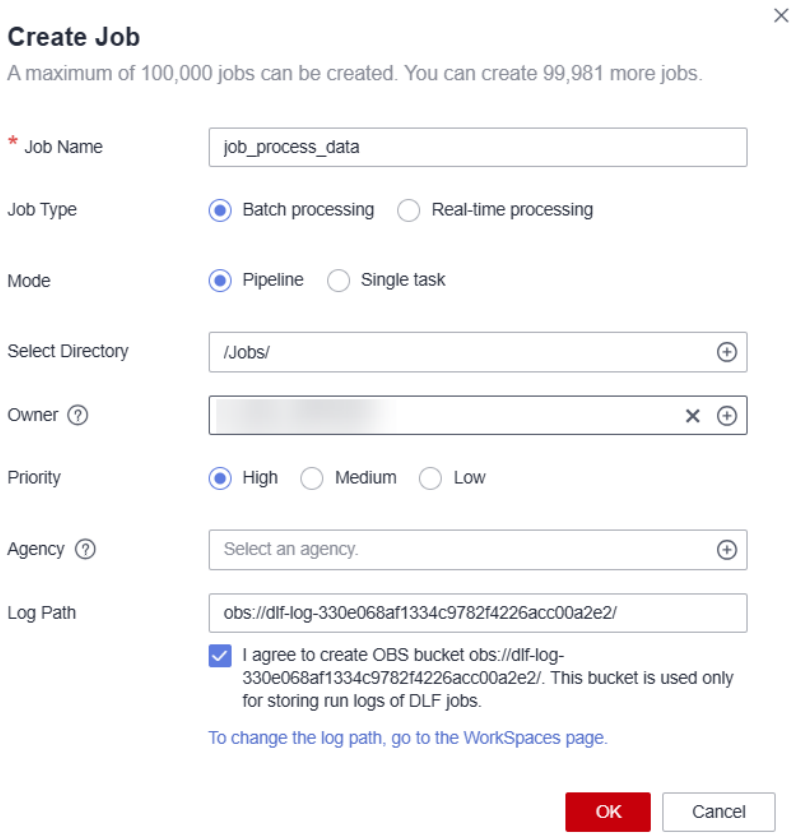
1. In the left navigation pane of DataArts Factory, choose **Data Development** > **Develop Job**.

Figure 2-35 Creating a job



2. Click **Create Job**. In the dialog box that appears, edit job information. In this example, the SQL job is named **job_process_data**.

   Figure 2-36 Editing job information



3. On the job development page, drag the DLI SQL node to the canvas and click the node to edit its properties.
   - **SQL or Script**: Select **SQL script** in this example. Then, select the script created in **Step 1** for **SQL script**.
   - **Database Name**: Select the database configured in the SQL script.
   - **Queue Name**: Select the SQL queue created in **Create an elastic resource pool and create a SQL queue within it**.
   - Environment variable: The DLI environment variable is optional.

     The description of the parameter added in this example is as follows:

     spark.sql.optimizer.dynamicPartitionPruning.enabled = true

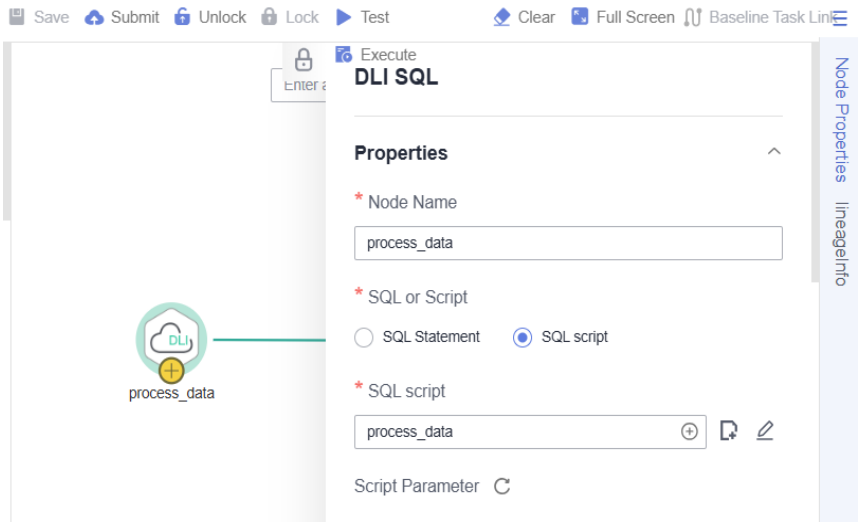
     - This parameter is used to control whether to enable dynamic partition pruning. Dynamic partition pruning can help reduce the amount of data that needs to be scanned and improve query performance when executing SQL queries.
     - When set to **true**, dynamic partition pruning is enabled. SQL automatically detects and deletes partitions that do not meet the WHERE clause conditions during query. This is useful for tables that have a large number of partitions.

- If SQL queries contain a large number of nested left join operations and the table has a large number of dynamic partitions, a large number of memory resources may be consumed during data parsing. As a result, the memory of the driver node is insufficient and there are frequent Full GCs.
- To avoid such issues, you can disable dynamic partition pruning by setting this parameter to **false**.

  However, disabling this optimization may reduce query performance. Once disabled, Spark does not automatically prune the partitions that do not meet the requirements.

For more property parameters, see **Parameters of DLI SQL nodes**.

Figure 2-37 Editing the properties of a DLI SQL node



4. Once the properties are edited, click **Save** to save the configuration.

----End

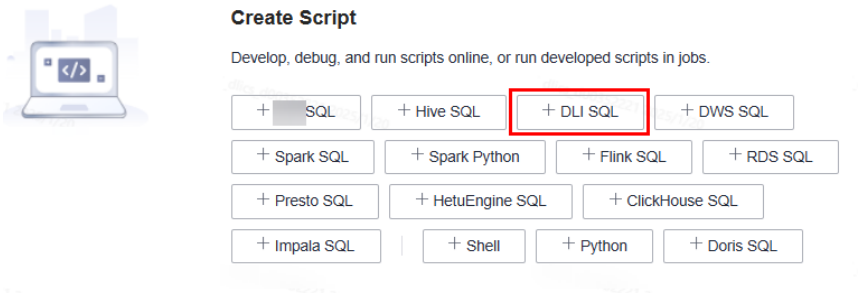
Step 3: Query and Analyze Sales Data

Develop a SQL script for analyzing and processing data.

This part describes how to submit a SQL script to analyze data.

1. In the left navigation pane of DataArts Factory, choose **Data Development** > **Develop Script**.
2. In the **Create Script** area on the right, click **+ DLI SQL**.

Figure 2-38 Creating a DLI SQL script



3. On the script editing page, enter sample code for analyzing data.

```
-- Query daily sales
SELECT transaction_id, productid, productname, quantity, dt FROM supermarket_db.analyze WHERE dt = '2024-11-01';
```

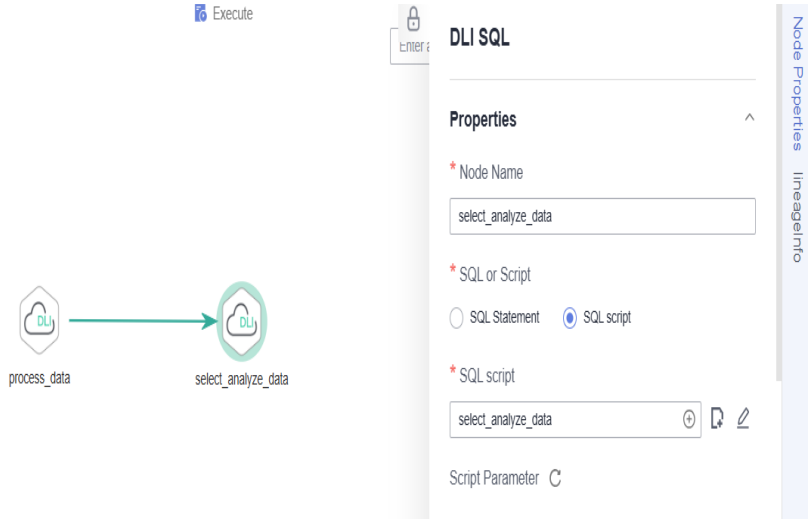
4. Click **Save** to save the SQL script. In this example, the script is named **select_analyze_data**.
5. Click **Submit** to execute the script.

Step 4: Orchestrate a Job

1. Create a DLI SQL node named **select_analyze_data** in the **job_process_data** job. Then, click the node to edit its properties.
   - **SQL or Script**: Select **SQL script** in this example. Then, select the script created in **Step 1** for **SQL script**.
   - **Database Name**: Select the database configured in the SQL script.
   - **Queue Name**: Select the SQL queue created in **Create an elastic resource pool and create a SQL queue within it**.

   For more property parameters, see **Parameters of DLI SQL nodes**.

Figure 2-39 Editing the properties of a DLI SQL node



2. Once the properties are edited, click **Save** to save the configuration.
3. Orchestrate the two nodes into a pipeline. DataArts Studio executes all nodes based on the orchestrated pipeline sequence. Then, click **Save** and **Submit** in the upper left corner.

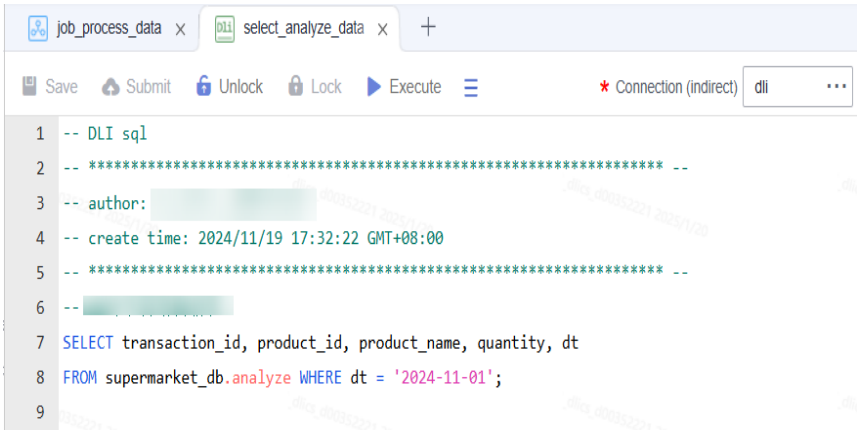
Step 5: Test Job Running

After orchestrating the job, click **Test** to test it.

After testing the job, open the **select_analyze_data** SQL script file and click **Execute** to query and analyze sales details.

If query results meet your expectation, go to **Step 6: Set Periodic Job Scheduling** to set periodic job scheduling.

Figure 2-40 Running the select_analyze_data script

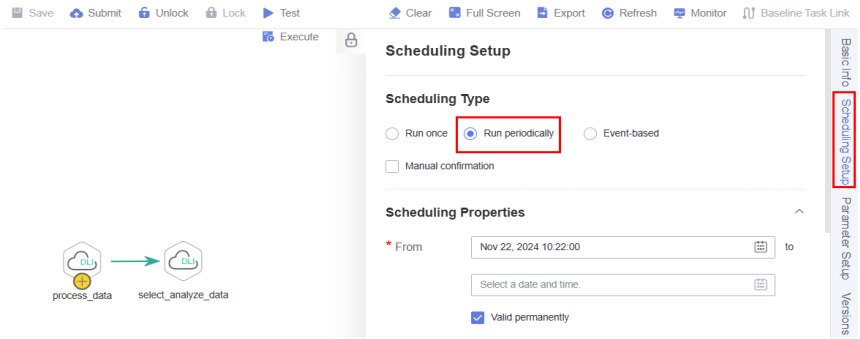


Step 6: Set Periodic Job Scheduling

1. In the left navigation pane of DataArts Factory, choose **Data Development** > **Develop Job**.
2. Double-click **job_process_data**.
3. Click the **Scheduling Setup** tab on the right.
4. Select **Run periodically** and set scheduling properties.

   In this example, the job's scheduling policy starts at 10:15:00 on November 22, 2024. The first scheduled time is 10:20:00 on the same day. With a daily scheduling interval, the job automatically runs at 10:20:00 a.m. each day, executing the nodes based on the orchestrated pipeline sequence.

   Figure 2-41 Configuring job scheduling



5. Click **Save**, **Submit**, and **Execute** in sequence to complete periodic scheduling configuration.

For more job scheduling settings, see **Setting Up Scheduling for a Job**.

Related Operations

- **Configure job monitoring.**

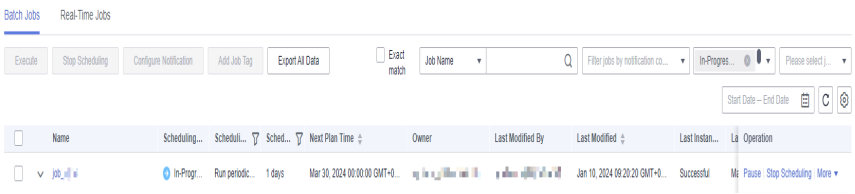
  DataArts Studio monitors the status of batch jobs.

  This type of job is a pipeline that consists of one or more nodes and is scheduled as a whole.

  In the left navigation pane of DataArts Factory, choose **Monitoring** > **Job Monitoring**. On the displayed **Batch Jobs** tab, view the scheduling status, scheduling interval, and scheduling start time of batch jobs.

  For details, see **Monitoring a Batch Job**.

  Figure 2-42 Configuring job monitoring

- **Configure instance monitoring.**

  Each time a job is executed, a job instance record is generated.

  In the left navigation pane of DataArts Factory, choose **Monitoring** > **Monitor Instance**. On the instance monitoring page that appears, you can view job instance information and perform more operations on the instances as needed.

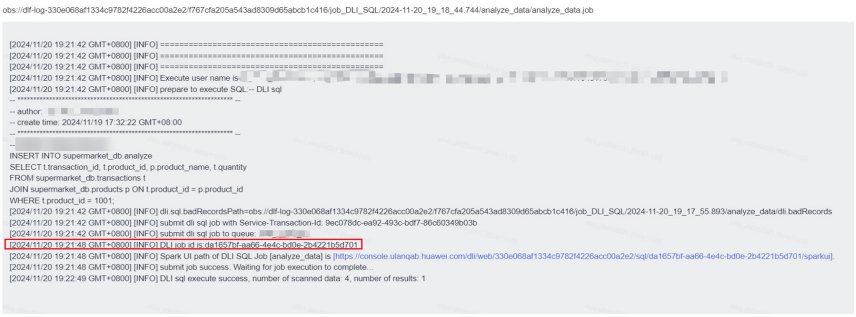
  For more information, see **Instance Monitoring**.

FAQ

- **If a DataArts Studio job fails, and the logs provided by DataArts Studio are not detailed enough, what should I do? Where can I find more specific logs?**

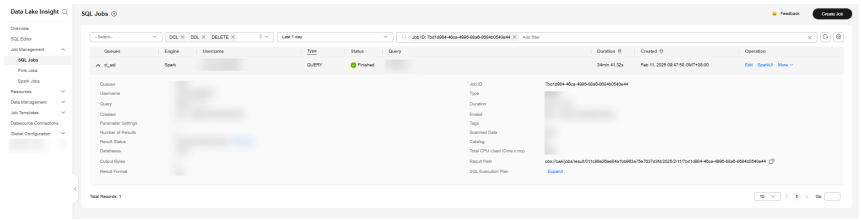
  You can locate the DLI job ID through the logs provided by DataArts Studio, and then find the specific job in the DLI console using the DLI job ID.

  Figure 2-43 Monitoring log file

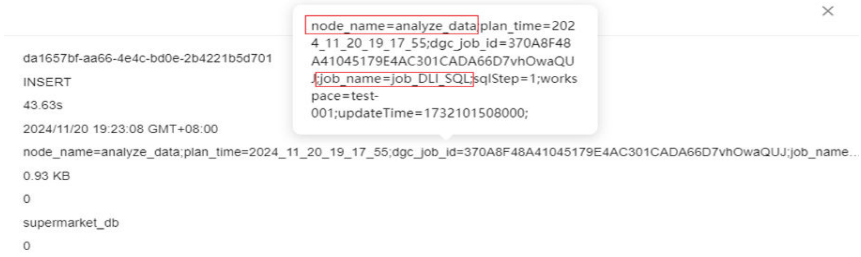
  Once you find the specific job in the DLI console, click archived logs to view detailed logs.

Figure 2-44 Entering the job ID



You can also search for the job in the DLI console using the node name or job name provided by DataArts Studio.

Figure 2-45 Node name or job name



- **What should I do if I encounter permission errors when running complex DLI jobs?**

  DLI needs to work with other cloud services. You must grant DLI basic operation permissions of these services so that DLI can access them and perform resource O&M operations on your behalf.

  For more information, see **Configuring DLI Agency Permissions**.

2.3 Calling UDFs in Spark SQL Jobs

Scenario

DLI allows you to use Hive user-defined functions (UDFs) to query data. UDFs take effect only on a single row of data and are applicable to inserting and deleting a single data record.

Constraints

- To perform UDF-related operations on DLI, you need to create a SQL queue instead of using the default queue.
- When UDFs are used across accounts, other users, except the user who creates them, need to be authorized before using the UDF. The authorization operations are as follows:

  Log in to the DLI console and choose **Data Management** > **Package Management**. On the displayed page, select your UDF Jar package and click **Manage Permissions** in the **Operation** column. On the permission management page, click **Grant Permission** in the upper right corner and select the required permissions.

- If you use a static class or interface in a UDF, add **try catch** to capture exceptions. Otherwise, package conflicts may occur.

Environment Preparations

Before you start, set up the development environment.

Table 2-2 Development environment

Item	Description
OS	Windows 7 or later
JDK	JDK 1.8.
IntelliJ IDEA	This tool is used for application development. The version of the tool must be 2019.1 or other compatible versions.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.

Development Process

The process of developing a UDF is as follows:

Figure 2-46 Development process

Table 2-3 Process description

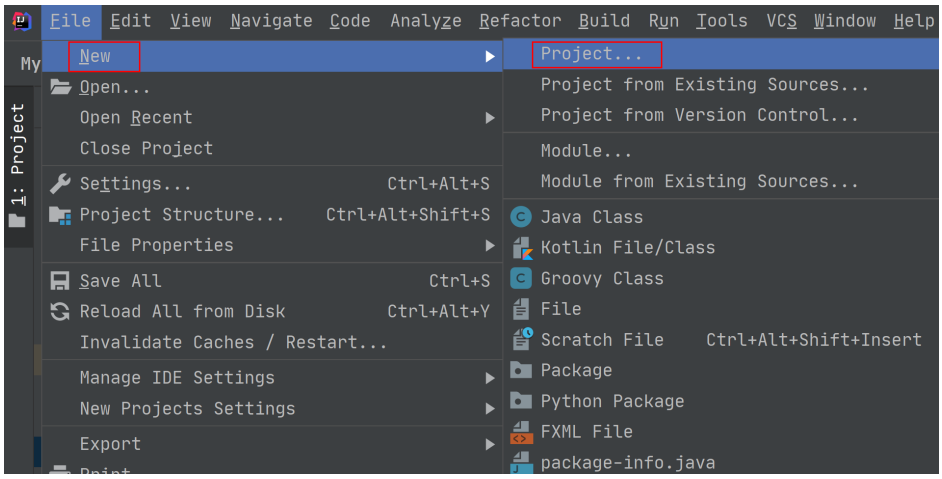
No.	Phase	Software Portal	Description
1	Create a Maven project and configure the POM file.	IntelliJ IDEA	Write UDF code by referring the steps in **Procedure**.
2	Write UDF code.		
3	Debug, compile, and pack the code into a Jar package.		
4	Upload the Jar package to OBS.	OBS console	Upload the UDF Jar file to an OBS directory.

No.	Phase	Software Portal	Description
5	Create the UDF on DLI.	DLI console	Create a UDF on the SQL job management page of the DLI console.
6	Verify and use the UDF on DLI.	DLI console	Use the UDF in your DLI job.

Procedure

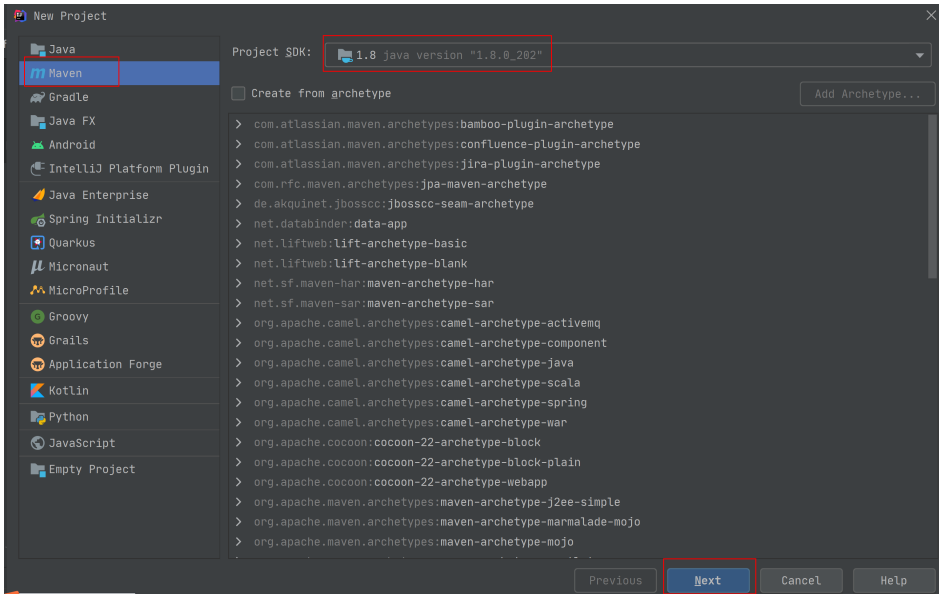
1. Create a Maven project and configure the POM file. This step uses IntelliJ IDEA 2020.2 as an example.
   a. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

      Figure 2-47 Creating a project

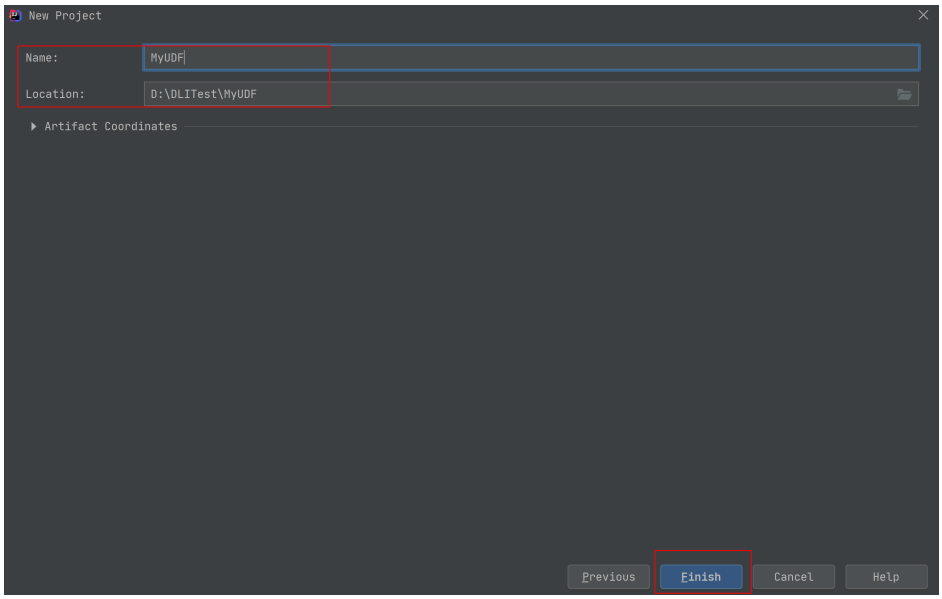
   b. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

Figure 2-48 Choosing Maven



c. Set the project name, configure the storage path, and click **Finish**.

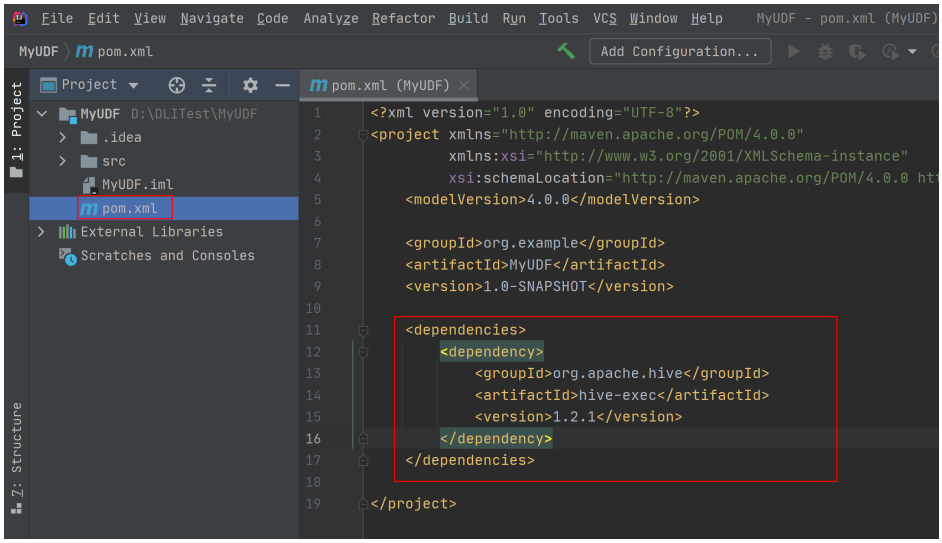
Figure 2-49 Creating a project



d. Add the following content to the **pom.xml** file.

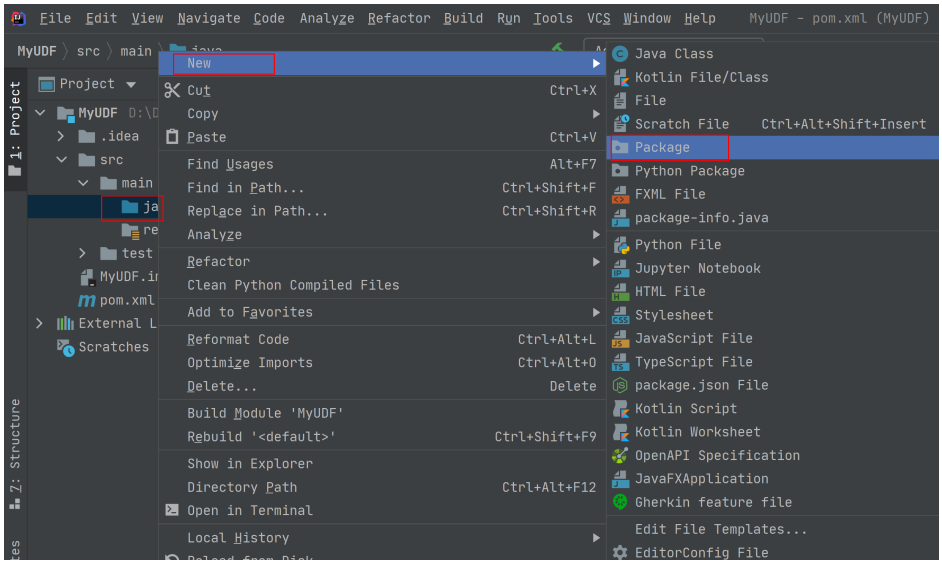
```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```


Figure 2-50 Adding configurations to the POM file



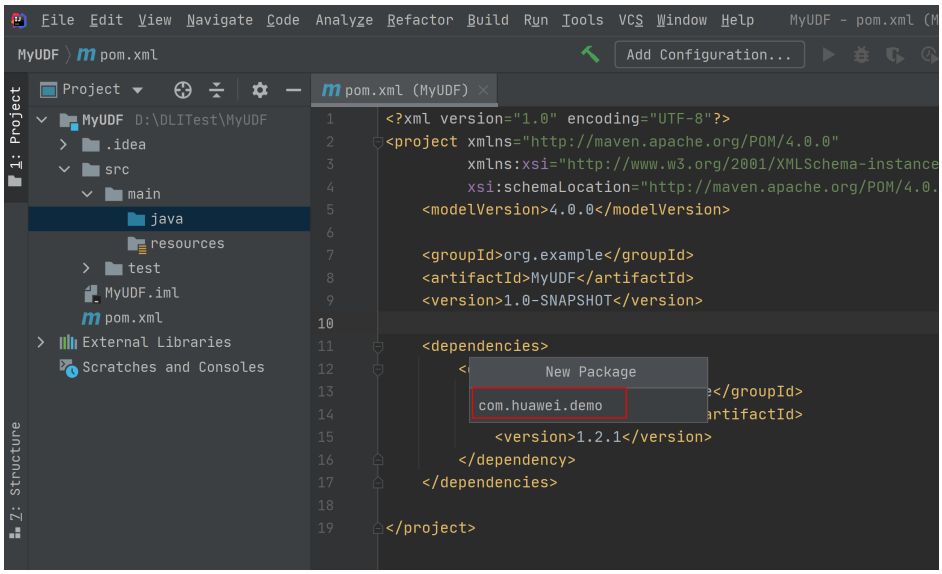
e. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

Figure 2-51 Creating a package and a class file



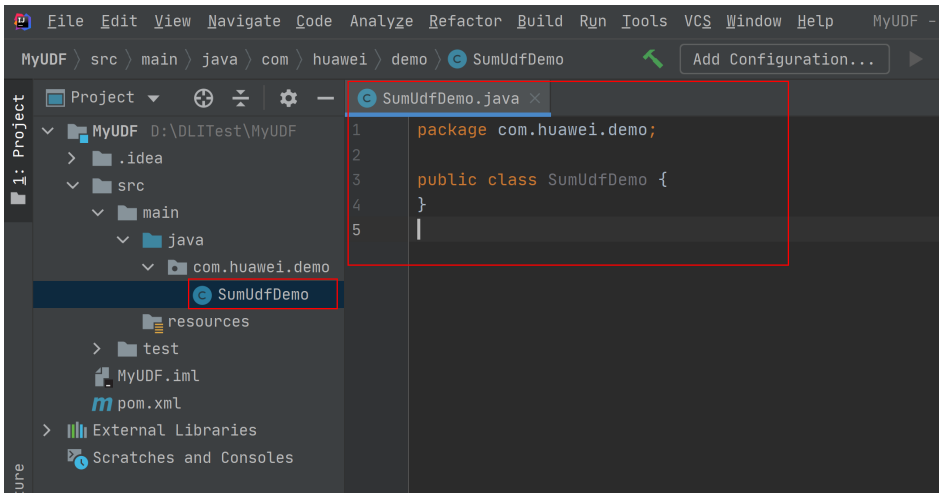
Set the package name as you need. In this example, set **Package** to **com.huawei.demo**. Then, press **Enter**.

Figure 2-52 Customizing a package



Create a Java Class file in the package path. In this example, the Java Class file is **SumUdfDemo**.

Figure 2-53 Creating a Java class file



2. Write UDF code.
   a. The UDF must inherit **org.apache.hadoop.hive.ql.exec.UDF**.
   b. You must implement the **evaluate** function, which can be reloaded.

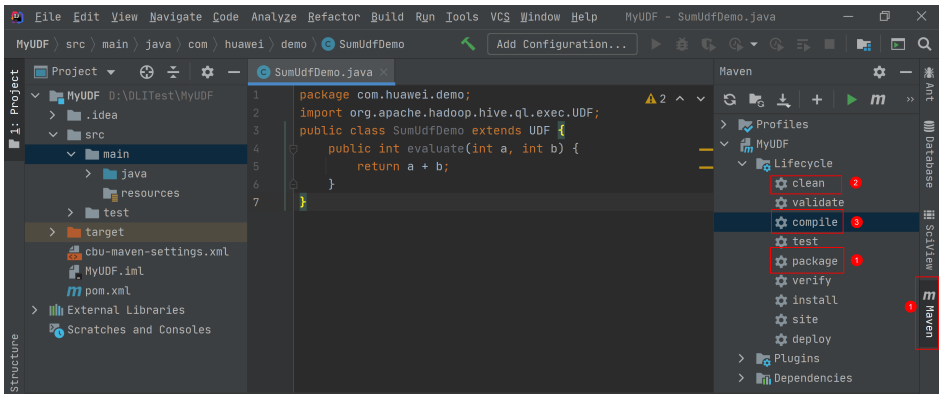
   For details about how to implement the UDF, see the following sample code:

```
package com.huawei.demo;
import org.apache.hadoop.hive.ql.exec.UDF;
 public class SumUdfDemo extends UDF {
   public int evaluate(int a, int b) {
    return a + b;
  }
 }
```

3. Use IntelliJ IDEA to compile the code and pack it into the JAR package.
   a. Click **Maven** in the tool bar on the right, and click **clean** and **compile** to compile the code.

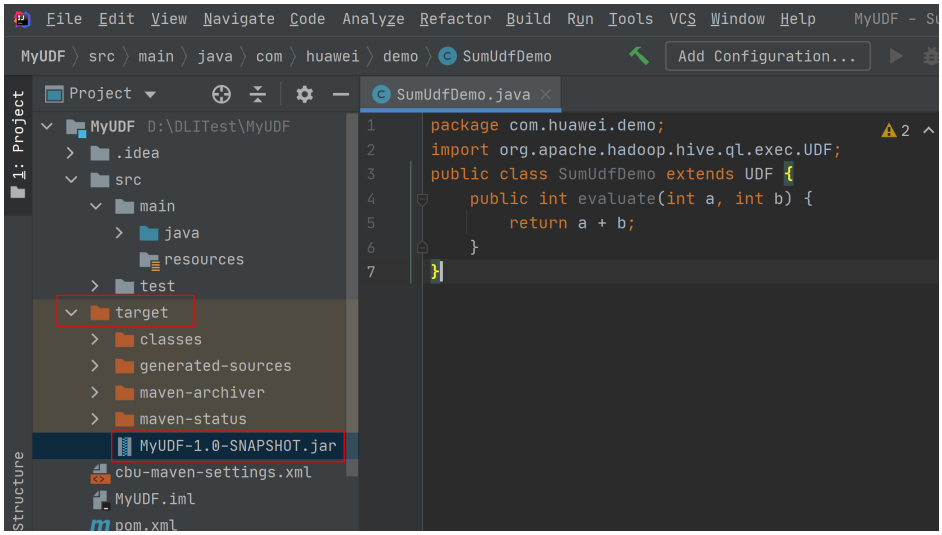
After the compilation is successful, click **package**.

Figure 2-54 Compiling and packaging



The generated JAR package is stored in the **target** directory. In this example, **MyUDF-1.0-SNAPSHOT.jar** is stored in **D:\DLITest\MyUDF\target**.

Figure 2-55 Generating a JAR file



4. Log in to the OBS console and upload the file to the OBS path.

   NOTE

   The region of the OBS bucket to which the Jar package is uploaded must be the same as the region of the DLI queue. Cross-region operations are not allowed.

5. (Optional) Upload the file to DLI for package management.

   a. Log in to the DLI management console and choose **Data Management** > **Package Management**.

   b. On the **Package Management** page, click **Create** in the upper right corner.

   c. In the **Create Package** dialog, set the following parameters:

      i. **Type**: Select **JAR**.

ii. **OBS Path**: Specify the OBS path for storing the package.

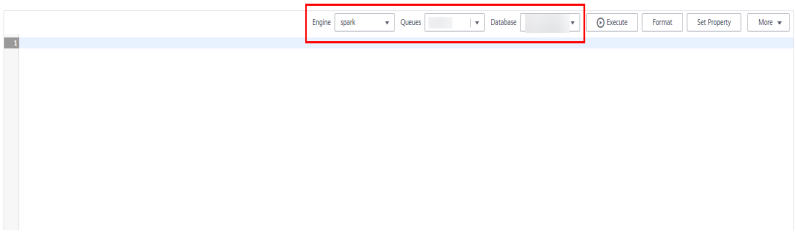
iii. Set **Group** and **Group Name** as required for package identification and management.

d. Click **OK**.

6. Create the UDF on DLI.

a. Log in to the DLI console, choose **SQL Editor**. Set **Engine** to **spark**, and select the created SQL queue and database.

Figure 2-56 Selecting the queue and database



b. In the SQL editing area, run the following statement to create a UDF and click **Execute**.

```
CREATE FUNCTION TestSumUDF AS 'com.huawei.demo.SumUdfDemo' using jar 'obs://dli-test-obs01/MyUDF-1.0-SNAPSHOT.jar';
```

7. Restart the original SQL queue for the added function to take effect.

a. Log in to the DLI console and choose **Queue Management** from the navigation pane. In the **Operation** column of the SQL queue job, click **Restart**.

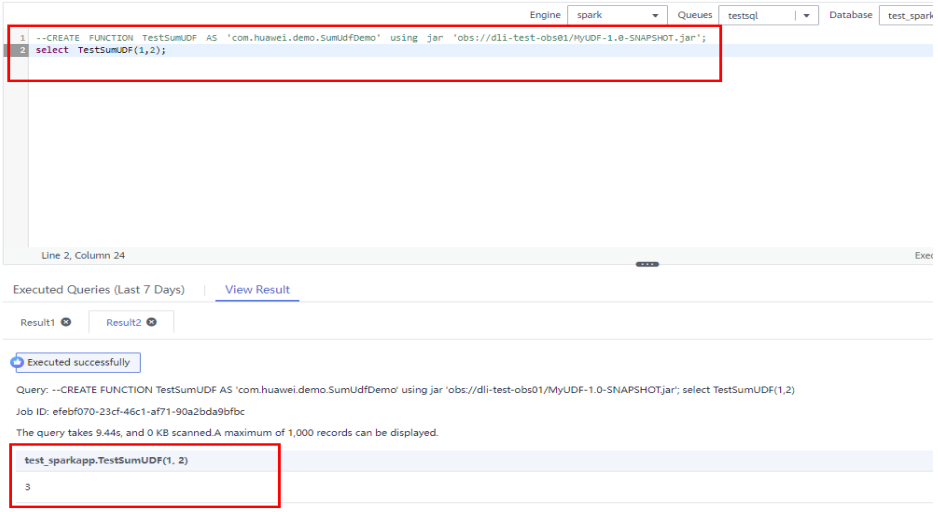
b. In the **Restart** dialog box, click **OK**.

8. Call the UDF.

Use the UDF created in **6** in the SELECT statement as follows:

```
select TestSumUDF(1,2);
```

Figure 2-57 Execution result



9. (Optional) Delete the UDF.

If the UDF is no longer used, run the following statement to delete it:

```
Drop FUNCTION TestSumUDF;
```


2.4 Calling UDTFs in Spark SQL Jobs

Scenario

You can use Hive User-Defined Table-Generating Functions (UDTF) to customize table-valued functions. Hive UDTFs are used for the one-in-multiple-out data operations. UDTF reads a row of data and output multiple values.

Constraints

- To perform UDTF-related operations on DLI, you need to create a SQL queue instead of using the default queue.
- When UDTFs are used by multiple accounts, other users, except the user who creates them, need to be authorized before using the UDTF. The authorization operations are as follows:

  Log in to the DLI console and choose **Data Management** > **Package Management**. On the displayed page, select your UDTF Jar package and click **Manage Permissions** in the **Operation** column. On the permission management page, click **Grant Permission** in the upper right corner and select the required permissions.
- If you use a static class or interface in a UDF, add **try catch** to capture exceptions. Otherwise, package conflicts may occur.

Environment Preparations

Before you start, set up the development environment.

Table 2-4 Development environment

Item	Description
OS	Windows 7 or later
JDK	JDK 1.8.
IntelliJ IDEA	This tool is used for application development. The version of the tool must be 2019.1 or other compatible versions.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.

Development Process

The process of developing a UDTF is as follows:

Figure 2-58 Development process

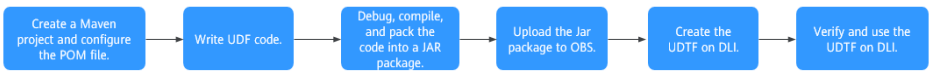


Table 2-5 Process description

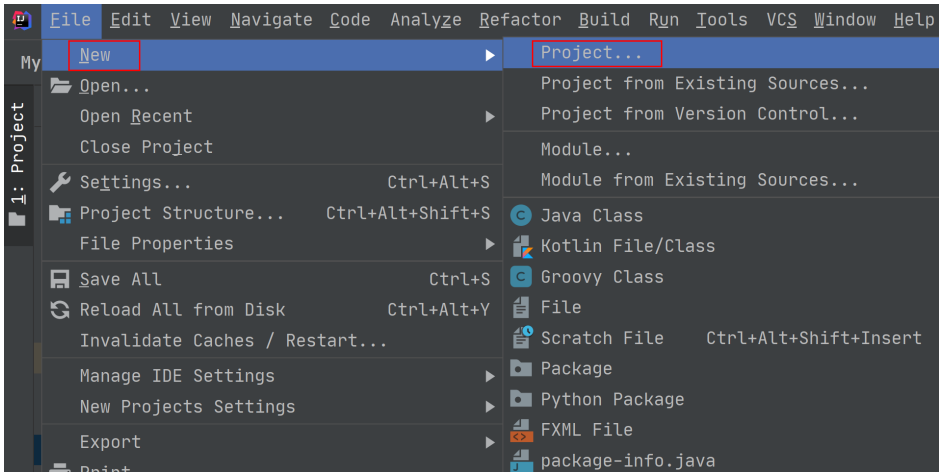
No.	Phase	Software Portal	Description
1	Create a Maven project and configure the POM file.	IntelliJ IDEA	Write UDTF code by referring the steps in **Procedure**.
2	Write UDTF code.		
3	Debug, compile, and pack the code into a Jar package.		
4	Upload the Jar package to OBS.	OBS console	Upload the UDTF Jar file to an OBS directory.
5	Create the UDTF on DLI.	DLI console	Create a UDTF on the SQL job management page of the DLI console.
6	Verify and use the UDTF on DLI.	DLI console	Use the UDTF in your DLI job.

Procedure

1. Create a Maven project and configure the POM file. This step uses IntelliJ IDEA 2020.2 as an example.

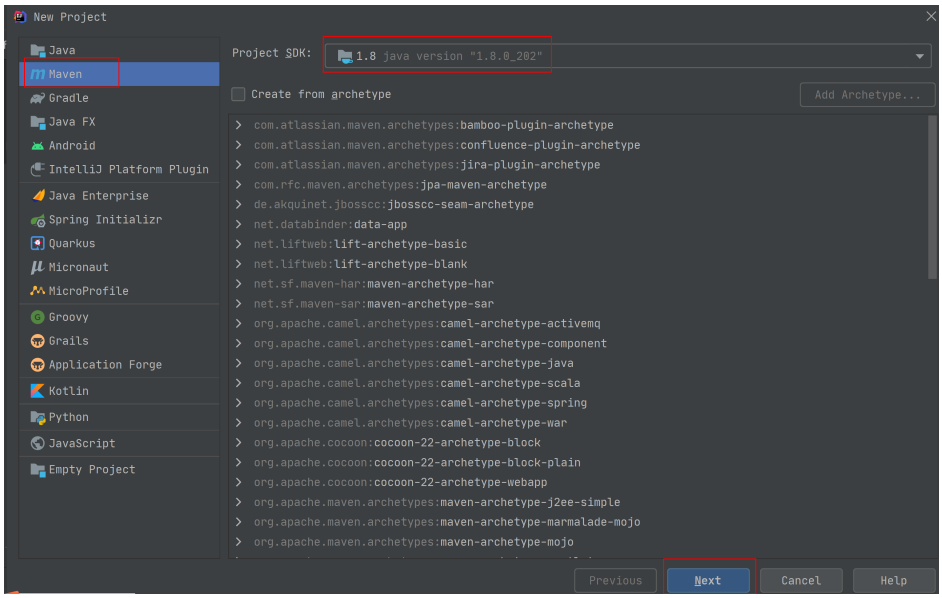
   a. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

   Figure 2-59 Creating a project



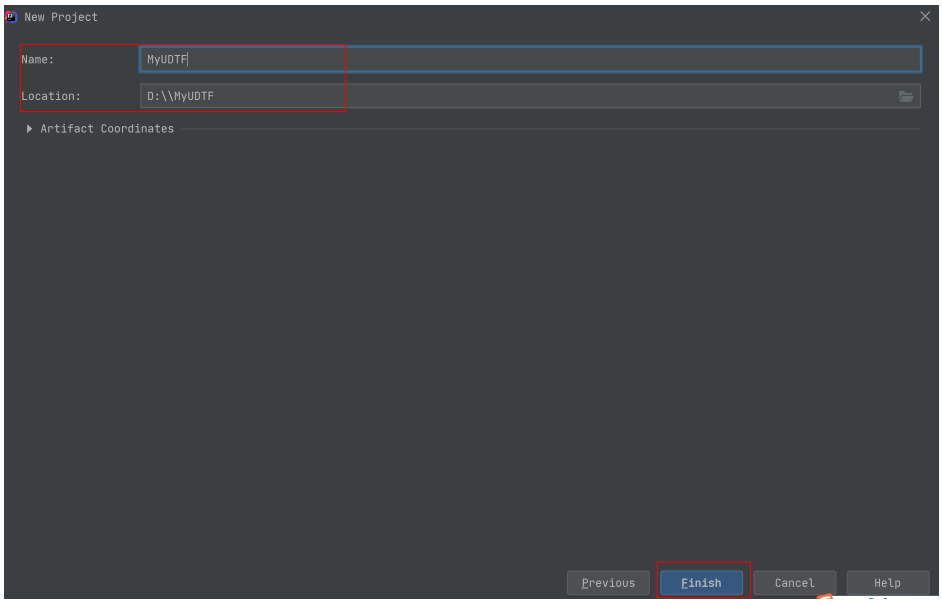
b. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

Figure 2-60 Choosing Maven



c. Set the project name, configure the storage path, and click **Finish**.

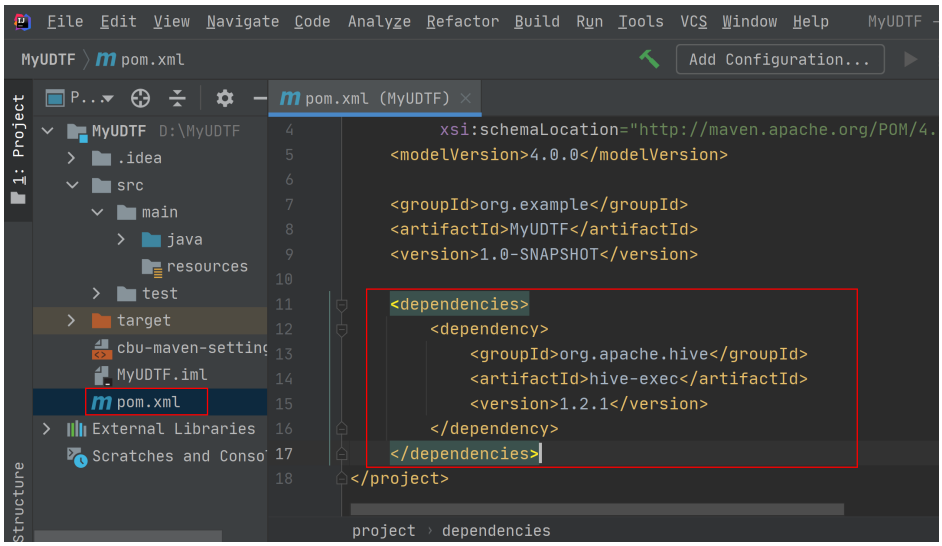
Figure 2-61 Creating a project



d. Add the following content to the **pom.xml** file.

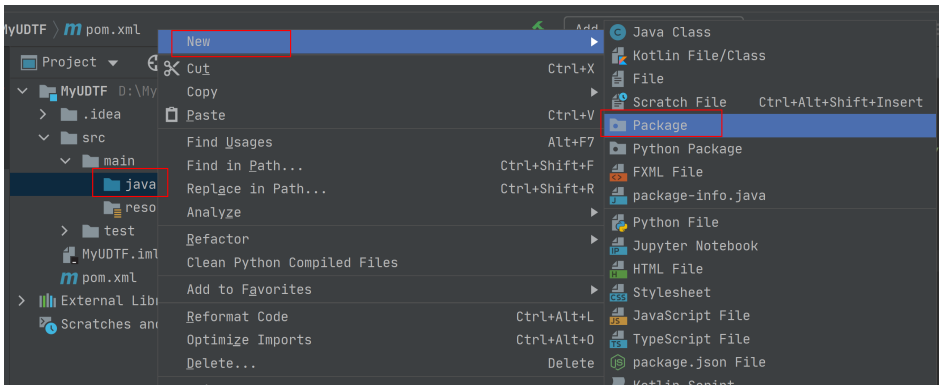
```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```


Figure 2-62 Adding configurations to the POM file



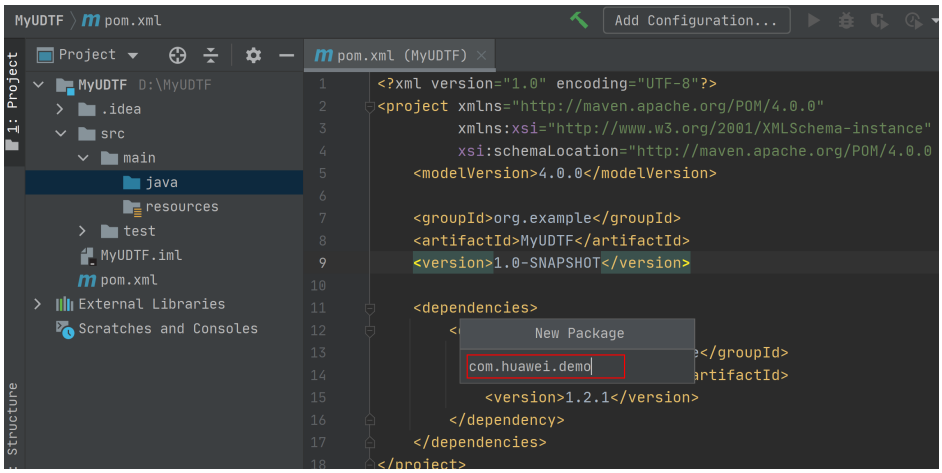
e. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

Figure 2-63 Creating a package and a class file



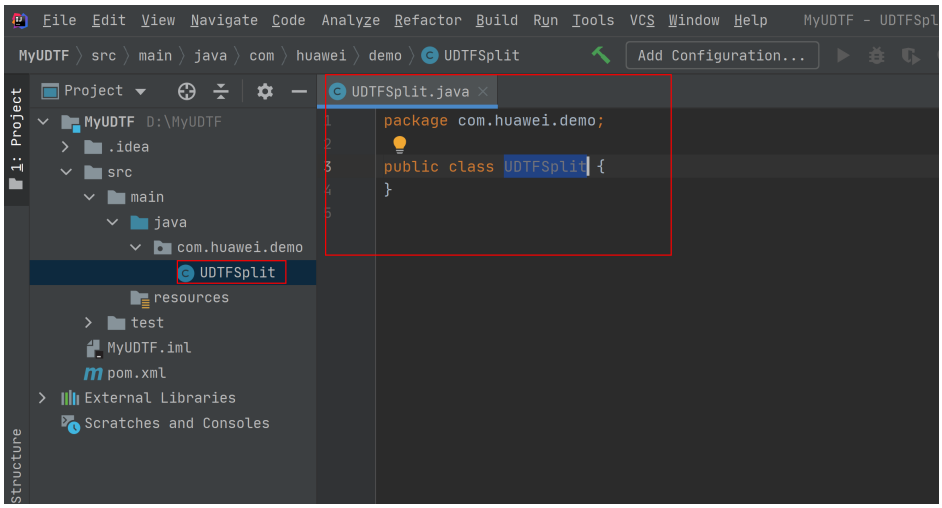
Set the package name as you need. In this example, set **Package** to **com.huawei.demo**. Then, press **Enter**.

Figure 2-64 Customizing a package



Create a Java Class file in the package path. In this example, the Java Class file is **UDTFSplit**.

Figure 2-65 Creating a Java class file



2. Write UDTF code. For sample code, see **Sample Code**.

   The UDTF class must inherit **org.apache.hadoop.hive.ql.udf.generic.GenericUDTF** to implement the **initialize**, **process**, and **close** methods.

   a. Call the **initialize** method in the UDTF. This method returns the information about the returned data rows of the UDTF, such as the number and type.

   b. Call the **process** method to process data. Each time **forward()** is called in the **process** method, a row is generated.

      If multiple columns are generated, you can put the values in an array and pass the array to the **forward()** function.

```
public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }
```

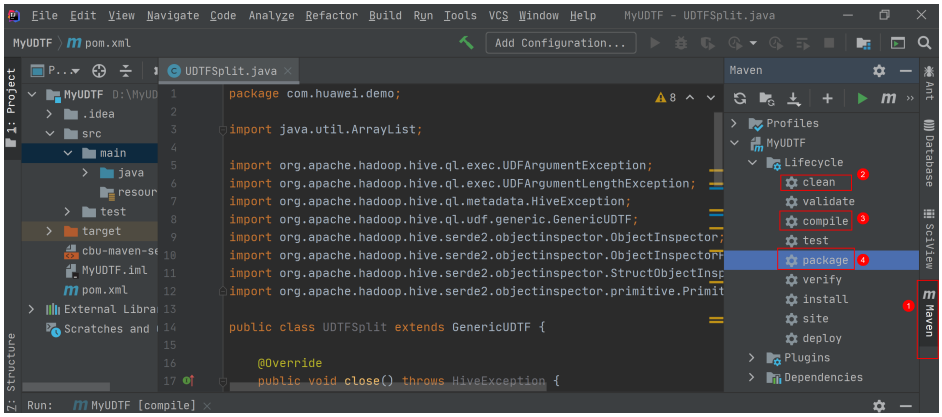
   c. Call the **close** method to clear methods that need to be closed.

3. Use IntelliJ IDEA to compile the code and pack it into the JAR package.

   a. Click **Maven** in the tool bar on the right, and click **clean** and **compile** to compile the code.

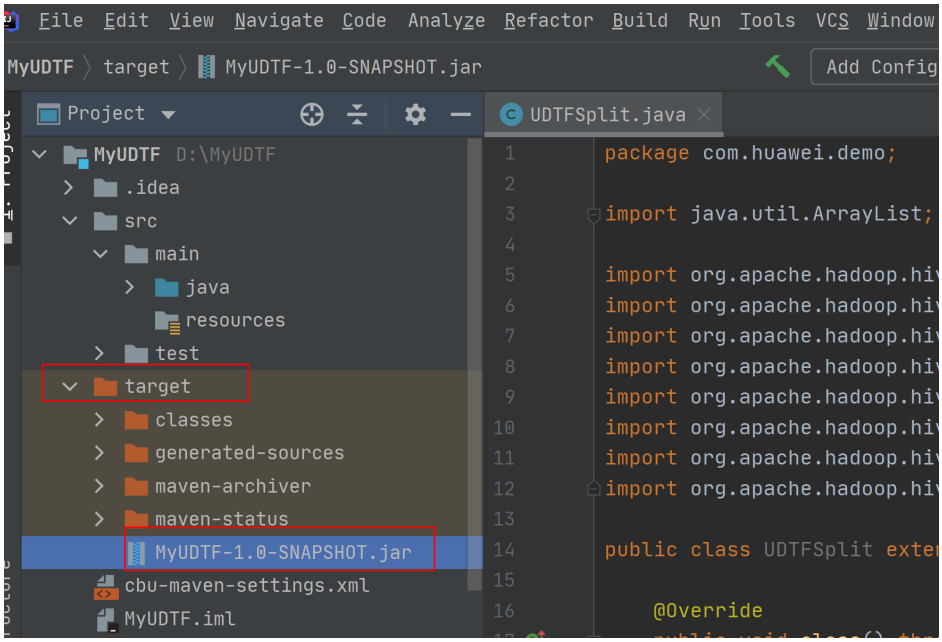
After the compilation is successful, click **package**.

Figure 2-66 Compiling and packaging



The generated JAR package is stored in the **target** directory. In this example, **MyUDTF-1.0-SNAPSHOT.jar** is stored in **D:\MyUDTF\target**.

Figure 2-67 Generating a JAR file



4. Log in to the OBS console and upload the file to the OBS path.

   NOTE

   The region of the OBS bucket to which the Jar package is uploaded must be the same as the region of the DLI queue. Cross-region operations are not allowed.

5. (Optional) Upload the file to DLI for package management.

   a. Log in to the DLI management console and choose **Data Management** > **Package Management**.

   b. On the **Package Management** page, click **Create** in the upper right corner.

c. In the **Create Package** dialog, set the following parameters:

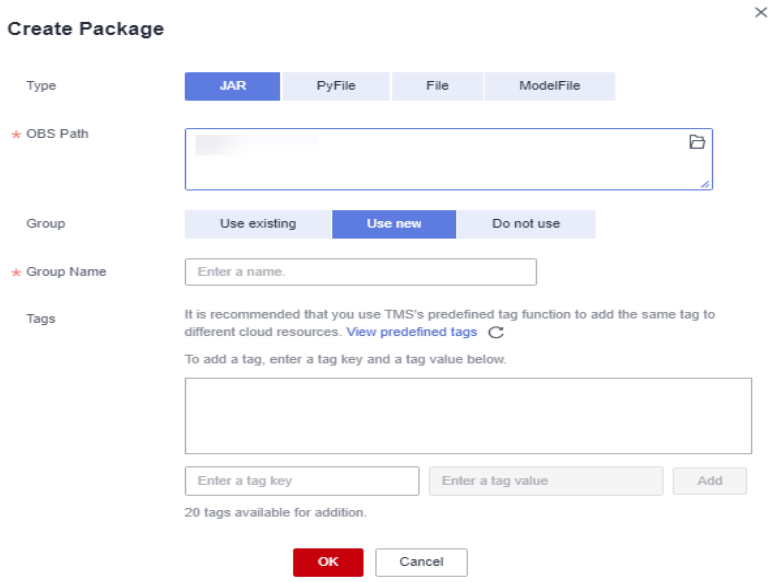
i. **Type**: Select **JAR**.

ii. **OBS Path**: Specify the OBS path for storing the package.

iii. Set **Group** and **Group Name** as required for package identification and management.

d. Click **OK**.

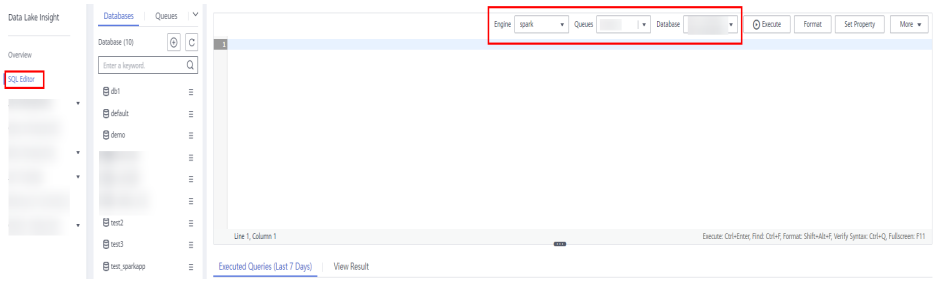
Figure 2-68 Creating a package



6. Create the UDTF on DLI.

a. Log in to the DLI console, choose **SQL Editor**. Set **Engine** to **spark**, and select the created SQL queue and database.

Figure 2-69 Selecting the queue and database



b. In the SQL editing area, enter the path of the JAR file to be uploaded to create a UDTF and click **Execute**.

```
CREATE FUNCTION mytestsplit AS 'com.huawei.demo.UDTFSplit' using jar 'obs://dli-test-obs01/MyUDTF-1.0-SNAPSHOT.jar';
```

7. Restart the original SQL queue for the added function to take effect.

a. Log in to the DLI management console and choose **Resources** > **Queue Management** from the navigation pane. In the **Operation** column of the SQL queue job, click **Restart**.

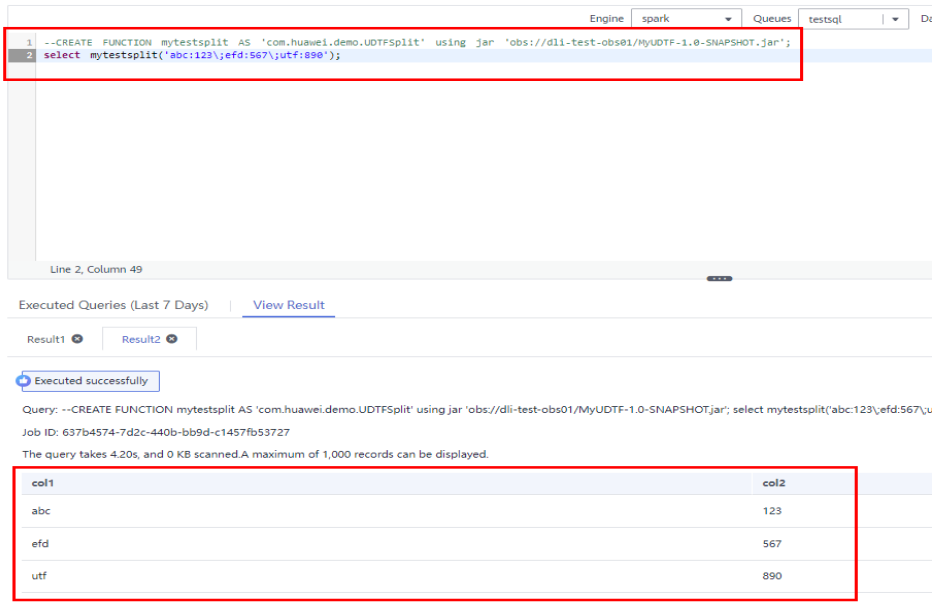
b. In the **Restart** dialog box, click **OK**.

8. Verify and use the UDTF on DLI.

   Use the UDTF created in **6** in the SELECT statement as follows:

   ```
   select mytestsplit('abc:123\;efd:567\;utf:890');
   ```

   Figure 2-70 Execution result



9. (Optional) Delete the UDTF.

   If this function is no longer used, run the following statement to delete the function:

   ```
   Drop FUNCTION mytestsplit;
   ```

Sample Code

The complete **UDTFSplit.java** code is as follows:

```
import java.util.ArrayList;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentLengthException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;

public class UDTFSplit extends GenericUDTF {

    @Override
    public void close() throws HiveException {
        // TODO Auto-generated method stub

    }

    @Override
    public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
```



```
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {
        if (args.length != 1) {
            throw new UDFArgumentLengthException("ExplodeMap takes only one argument");
        }
        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE) {
            throw new UDFArgumentException("ExplodeMap takes string as a parameter");
        }

        ArrayList<String> fieldNames = new ArrayList<String>();
        ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();
        fieldNames.add("col1");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        fieldNames.add("col2");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);
    }

}
```

2.5 Calling UDAFs in Spark SQL Jobs

Scenario

DLI allows you to use a Hive User Defined Aggregation Function (UDAF) to process multiple rows of data. Hive UDAF is usually used together with groupBy. It is equivalent to SUM() and AVG() commonly used in SQL and is also an aggregation function.

Constraints

- To perform UDAF-related operations on DLI, you need to create a SQL queue instead of using the default queue.
- When UDAFs are used across accounts, other users, except the user who creates them, need to be authorized before using the UDAF.

  To grant required permissions, log in to the DLI console and choose **Data Management** > **Package Management**. On the displayed page, select your UDAF Jar package and click **Manage Permissions** in the **Operation** column. On the permission management page, click **Grant Permission** in the upper right corner and select the required permissions.
- If you use a static class or interface in a UDF, add **try catch** to capture exceptions. Otherwise, package conflicts may occur.

Environment Preparations

Before you start, set up the development environment.

Table 2-6 Development environment

Item	Description
OS	Windows 7 or later
JDK	JDK 1.8 (**Java downloads**).
IntelliJ IDEA	**IntelliJ IDEA** is used for application development. The version of the tool must be 2019.1 or later.
Maven	Basic configuration of the development environment. For details about how to get started, see **Downloading Apache Maven** and **Installing Apache Maven**. Maven is used for project management throughout the lifecycle of software development.

Development Process

The following figure shows the process of developing a UDAF.

Figure 2-71 Development process

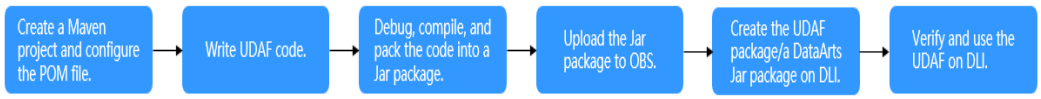


Table 2-7 Process description

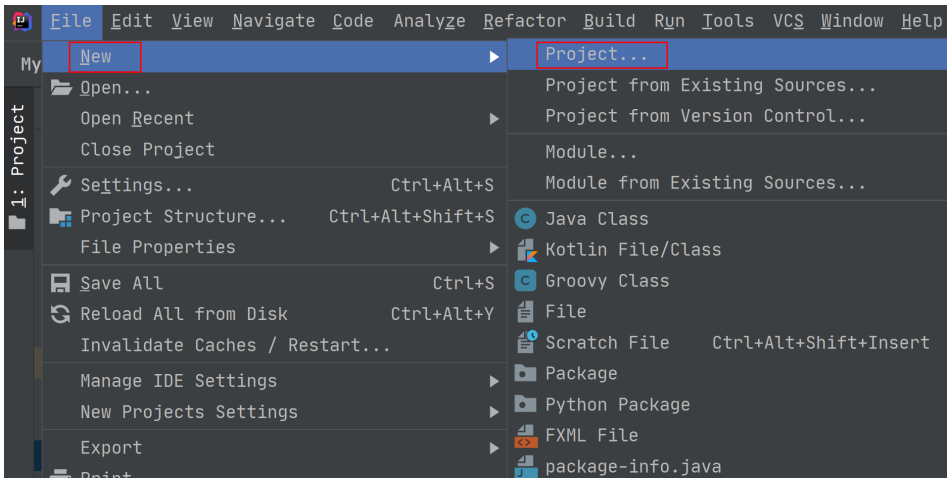
No.	Phase	Software Portal	Description
1	Create a Maven project and configure the POM file.	IntelliJ IDEA	Compile the UDAF function code by referring to the **Procedure** description.
2	Editing UDAF code		
3	Debug, compile, and pack the code into a Jar package.		
4	Upload the Jar package to OBS.	OBS console	Upload the UDAF Jar file to an OBS path.
5	Create a DLI package.	DLI console	Select the UDAF Jar file that has been uploaded to OBS for management.

No.	Phase	Software Portal	Description
6	Create a UDAF on DLI.	DLI console	Create a UDAF on the SQL job management page of the DLI console.
7	Verify and use the UDAF.	DLI console	Use the UDAF in your DLI job.

Procedure

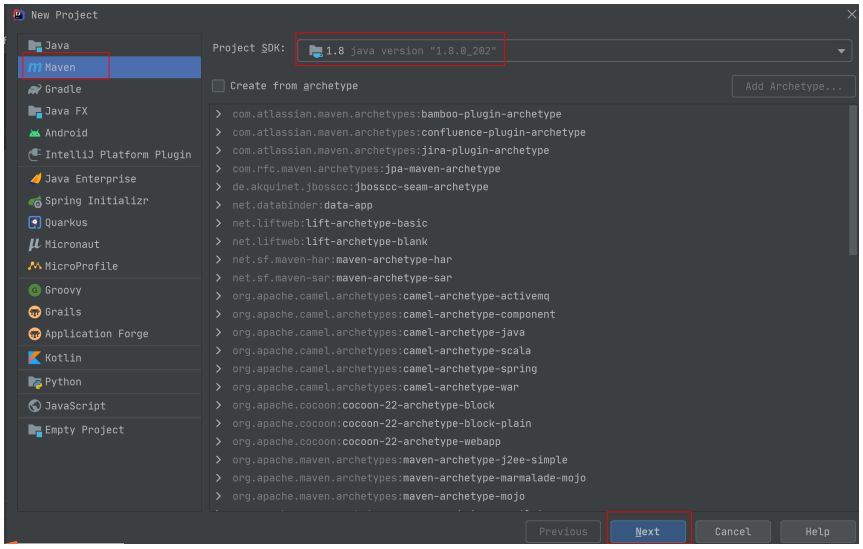
1. Create a Maven project and configure the POM file. This step uses IntelliJ IDEA 2020.2 as an example.
   a. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

      Figure 2-72 Creating a project

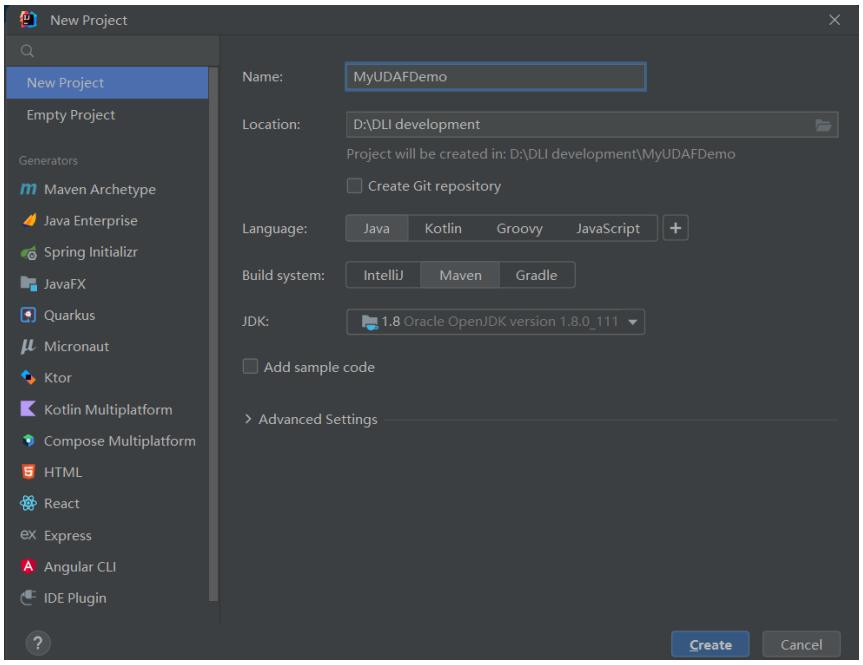
   b. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

      Figure 2-73 Configuring the project SDK

c. Specify the project name and the project path, and click **Create**. In the displayed page, click **Finish**.

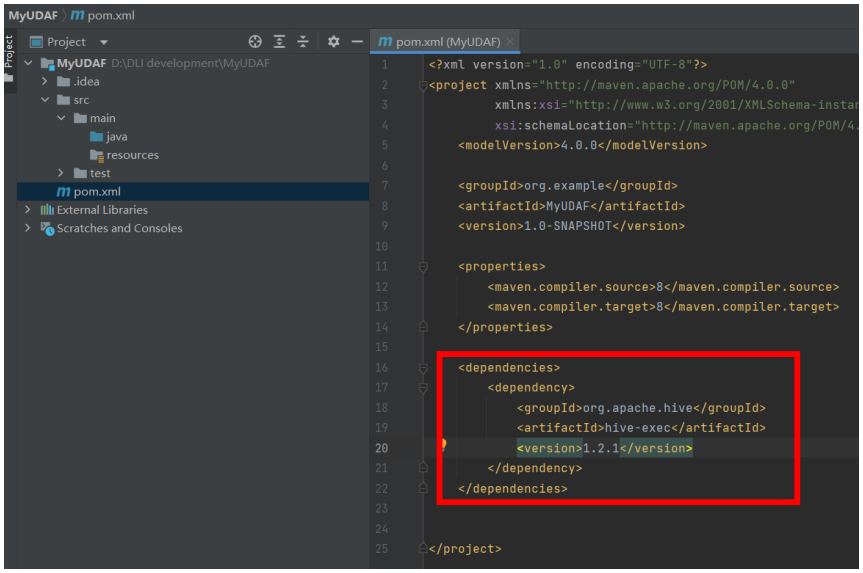
Figure 2-74 Setting project information



d. Add the following content to the **pom.xml** file.

```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

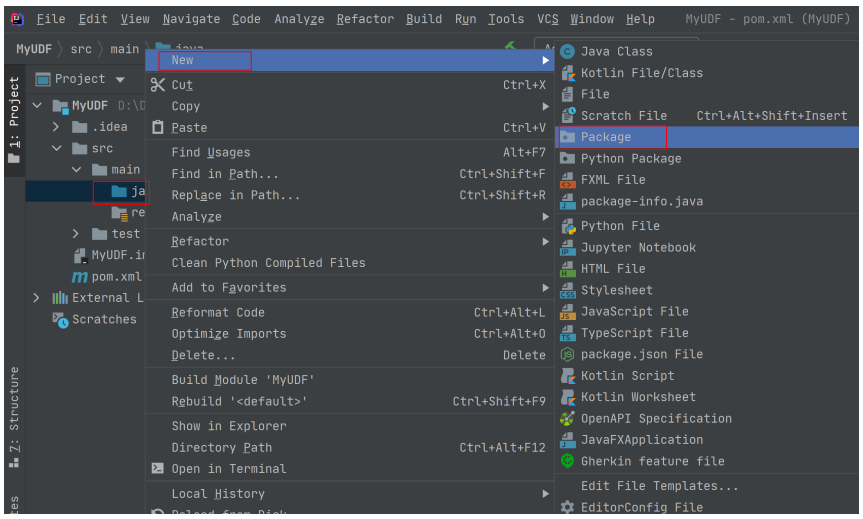
Figure 2-75 Adding configurations to the POM file



e. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

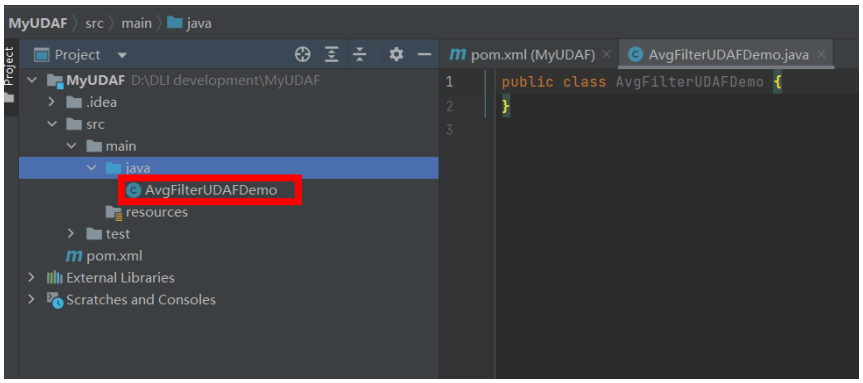
Set **Package** as required. In this example, set **Package** to **com.dli.demo**.

Figure 2-76 Creating a package



Create a Java Class file in the package path. In this example, the Java Class file is **AvgFilterUDAFDemo**.

Figure 2-77 Creating a class



2. Write UDAF code. Pay attention to the following requirements when you implement the UDAF:
   - The UDAF class must inherit from **org.apache.hadoop.hive.ql.exec.UDAF** and **org.apache.hadoop.hive.ql.exec.UDAFEvaluator** classes. The function class must inherit from the UDAF class, and the **Evaluator** class must implement the **UDAFEvaluator** interface.
   - The **Evaluator** class must implement the **init**, **iterate**, **terminatePartial**, **merge**, and **terminate** functions of **UDAFEvaluator**.
     - The **init** function overrides the **init** function of the **UDAFEvaluator** interface.
     - The **iterate** function receives input parameters for internal iteration.
     - The **terminatePartial** function has no parameter. It returns the data obtained after the **iterate** traversal is complete. **terminatePartial** is similar to Hadoop **Combiner**.

- The **merge** function receives the return values of **terminatePartial**.
- The **terminate** function returns the aggregated result.

For details about how to implement the UDAF, see the following sample code:

```
package com.dli.demo;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

/***
 * @jdk jdk1.8.0
 * @version 1.0
 ***/
public class AvgFilterUDAFDemo extends UDAF {

    /**
     * Defines the static inner class AvgFilter.
     */
    public static class PartialResult
    {
        public Long sum;
    }

    public static class VarianceEvaluator implements UDAFEvaluator {

        // Initializes the PartialResult object.
        private AvgFilterUDAFDemo.PartialResult partial;

        // Declares a VarianceEvaluator constructor that has no parameters.
        public VarianceEvaluator(){

            this.partial = new AvgFilterUDAFDemo.PartialResult();

            init();
        }

        /**
         * Initializes the UDAF, which is similar to a constructor.
         */
        @Override
        public void init() {

            // Sets the initial value of sum.
            this.partial.sum = 0L;
        }

        /**
         * Receives input parameters for internal iteration.
         * @param x
         * @return
         */
        public void iterate(Long x) {
            if (x == null) {
                return;
            }
            AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
            tmp9_6.sum = tmp9_6.sum | x;
        }

        /**
         * Returns the data obtained after the iterate traversal is complete.
         * terminatePartial is similar to Hadoop Combiner.
         * @return
         */
        public AvgFilterUDAFDemo.PartialResult terminatePartial()
        {
```

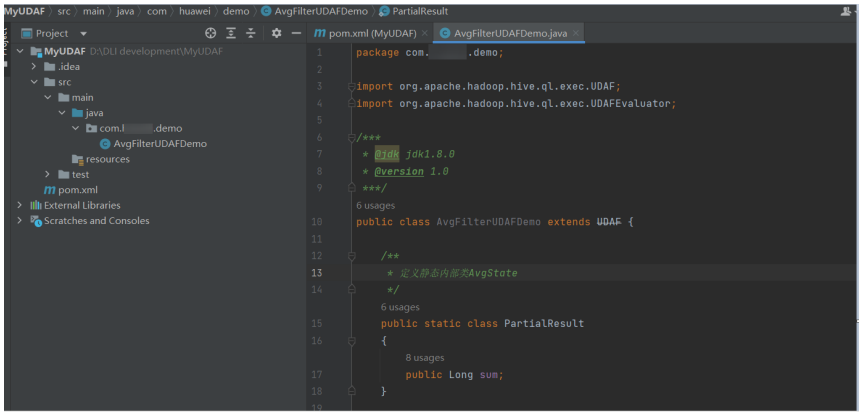


```
        return this.partial;
    }

    /**
     * Receives the return values of terminatePartial and merges the data.
     * @param
     * @return
     */
    public void merge(AvgFilterUDAFDemo.PartialResult pr)
    {
        if (pr == null) {
            return;
        }
        AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
        tmp9_6.sum = tmp9_6.sum | pr.sum;
    }

    /**
     * Returns the aggregated result.
     * @return
     */
    public Long terminate()
    {
        if (this.partial.sum == null) {
            return 0L;
        }
        return this.partial.sum;
    }
  }
}
```

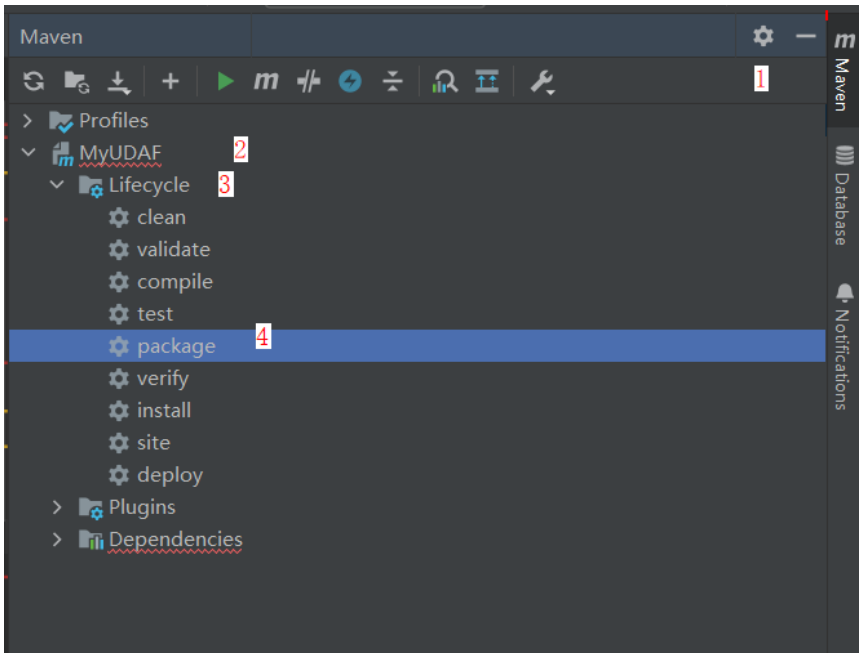
Figure 2-78 Editing UDAF code



3. Use IntelliJ IDEA to compile the code and pack it into the JAR package.
   a. Click **Maven** in the tool bar on the right, and click **clean** and **compile** to compile the code.

      After the compilation is successful, click **package**.

Figure 2-79 Exporting the Jar file



b. The generated JAR package is stored in the **target** directory. In this example, **MyUDAF-1.0-SNAPSHOT.jar** is stored in **D:\DLITest\MyUDAF\target**.

4. Log in to the OBS console and upload the file to the OBS path.

   NOTE

   The region of the OBS bucket to which the Jar package is uploaded must be the same as the region of the DLI queue. Cross-region operations are not allowed.

5. (Optional) Upload the file to DLI for package management.

   a. Log in to the DLI management console and choose **Data Management** > **Package Management**.

   b. On the **Package Management** page, click **Create** in the upper right corner.

   c. In the **Create Package** dialog, set the following parameters:

      - **Type**: Select **JAR**.
      - **OBS Path**: Specify the OBS path for storing the package.
      - Set **Group** and **Group Name** as required for package identification and management.

   d. Click **OK**.

6. Create the UDAF on DLI.

   a. Log in to the DLI management console and create a SQL queue and a database.

   b. Log in to the DLI console, choose **SQL Editor**. Set **Engine** to **spark**, and select the created SQL queue and database.

   c. In the SQL editing area, run the following statement to create a UDAF and click **Execute**.

> **NOTE**
>
> If the reloading function of the UDAF is enabled, the create statement changes.
>
> ```
> CREATE FUNCTION AvgFilterUDAFDemo AS 'com.dli.demo.AvgFilterUDAFDemo' using jar 'obs://dli-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
> ```
>
> Or
>
> ```
> CREATE OR REPLACE FUNCTION AvgFilterUDAFDemo AS 'com.dli.demo.AvgFilterUDAFDemo' using jar 'obs://dli-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
> ```

7. Restart the original SQL queue for the added function to take effect.
   a. Log in to the DLI management console and choose **Resources** > **Queue Management** from the navigation pane. In the **Operation** column of the SQL queue, click **Restart**.
   b. In the **Restart** dialog box, click **OK**.
8. Use the UDAF.

   Use the UDAF function created in **6** in the query statement:

   ```
   select AvgFilterUDAFDemo(real_stock_rate) AS show_rate FROM dw_ad_estimate_real_stock_rate limit 1000;
   ```
9. (Optional) Delete the UDAF.

   If the UDAF is no longer used, run the following statement to delete it:

   ```
   Drop FUNCTION AvgFilterUDAFDemo;
   ```


3 Flink Jobs

3.1 Stream Ecosystem

Built on Flink and Spark, the stream ecosystem is fully compatible with the open-source Flink, Storm, and Spark APIs. It is enhanced in features and improved in performance to provide easy-to-use DLI with low latency and high throughput.

The DLI stream ecosystem includes the cloud service ecosystem, open source ecosystem, and custom stream ecosystem.

- Cloud service ecosystems

  DLI can be interconnected with other services by using Stream SQLs. You can directly use SQL statements to read and write data from various cloud services, such as Data Ingestion Service (DIS), Object Storage Service (OBS), CloudTable Service (CloudTable), MapReduce Service (MRS), Relational Database Service (RDS), Simple Message Notification (SMN), and Distributed Cache Service (DCS).
- Open-source ecosystems

  After connections to other VPCs are established through VPC peering connections, you can access all data sources and output targets (such as Kafka, HBase, and Elasticsearch) supported by Flink and Spark in your dedicated DLI queues.
- Custom stream ecosystems

  You can compile code to obtain data from the desired cloud ecosystem or open-source ecosystem as the input data of Flink jobs.

3.2 Flink OpenSource SQL Jobs

3.2.1 Reading Data from Kafka and Writing Data to RDS

NOTICE

This guide provides reference for Flink 1.12 only.

Description

In this example, we aim to query information about top three most-clicked offerings in each hour from a set of real-time click data. Offerings' real-time click data will be sent to Kafka as the input source, and then the analysis result of Kafka data is to be output to RDS.

For example, enter the following sample data:

```
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:01:00", "product_id":"0002",
"product_name":"name1"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:02:00", "product_id":"0002",
"product_name":"name1"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:06:00", "product_id":"0004",
"product_name":"name2"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:10:00", "product_id":"0003",
"product_name":"name3"}
{"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:15:00", "product_id":"0005",
"product_name":"name4"}
{"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:16:00", "product_id":"0005",
"product_name":"name4"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:56:00", "product_id":"0004",
"product_name":"name2"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:05:00", "product_id":"0005",
"product_name":"name4"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:10:00", "product_id":"0006",
"product_name":"name5"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 09:13:00", "product_id":"0006",
"product_name":"name5"}
```

Expected output:

```
2021-03-24 08:00:00 - 2021-03-24 08:59:59,0002,name1,2
2021-03-24 08:00:00 - 2021-03-24 08:59:59,0004,name2,2
2021-03-24 08:00:00 - 2021-03-24 08:59:59,0005,name4,2
2021-03-24 09:00:00 - 2021-03-24 09:59:59,0006,name5,2
2021-03-24 09:00:00 - 2021-03-24 09:59:59,0005,name4,1
```

Prerequisites

1. You have created a DMS for Kafka instance.
   For details, see **Getting Started with DMS for Kafka**.

   CAUTION

   When you create the instance, do not enable **Kafka SASL_SSL**.

2. You have created an RDS for MySQL DB instance.
   In this example, the RDS for MySQL database version is 8.0.
   For details, see **Buying an RDS for MySQL DB Instance**.

Overall Development Process

Overall Process

Figure 3-1 Job development process

Step 1: Create a Queue

Step 2: Create a Kafka Topic

Step 3: Create an RDS Database and Table

Step 4: Create an Enhanced Datasource Connection

Step 5: Run a Job

Step 6: Send Data and Query Results

Step 1: Create a Queue

1. Log in to the DLI console. In the navigation pane on the left, choose **Resources** > **Queue Management**.
2. On the displayed page, click **Buy Queue** in the upper right corner.
3. On the **Buy Queue** page, set queue parameters as follows:
   - **Billing Mode**: .
   - **Region** and **Project**: Retain the default values.
   - **Name**: Enter a queue name.

     NOTE

     The queue name can contain only digits, letters, and underscores (_), but cannot contain only digits or start with an underscore (_). The name must contain 1 to 128 characters.

     The queue name is case-insensitive. Uppercase letters will be automatically converted to lowercase letters.
   - **Type**: Select **For general purpose**. Select the **Dedicated Resource Mode**.
   - **AZ Mode** and **Specifications**: Retain the default values.
   - **Enterprise Project**: Select **default**.
   - **Advanced Settings**: Select **Custom**.
   - **CIDR Block**: Specify the queue network segment. For example, **10.0.0.0/16**.

     CAUTION

     The CIDR block of a queue cannot overlap with the CIDR blocks of DMS Kafka and RDS for MySQL DB instances. Otherwise, datasource connections will fail to be created.
   - Set other parameters as required.
4. Click **Buy**. Confirm the configuration and click **Submit**.

Step 2: Create a Kafka Topic

1. On the Kafka management console, click an instance name on the **DMS for Kafka** page. Basic information of the Kafka instance is displayed.
2. Choose **Topics**. On the displayed page, click **Create Topic**. Configure the following parameters:

- Topic Name For this example, enter **testkafkatopic**.
- **Partitions**: Set the value to **1**.
- **Replicas**: Set the value to **1**.

Retain default values for other parameters.

Step 3: Create an RDS Database and Table

1. Log in to the RDS console. On the displayed page, locate the target MySQL DB instance and choose **More** > **Log In** in the **Operation** column.
2. On the displayed login dialog box, enter the username and password and click **Log In**.
3. On the **Databases** page, click **Create Database**. In the displayed dialog box, enter **testrdsdb** as the database name and retain default values of rest parameters. Then, click **OK**.
4. In the **Operation** column of row where the created database locates, click **SQL Window** and enter the following statement to create a table:

```
CREATE TABLE clicktop (
    `range_time` VARCHAR(64) NOT NULL,
    `product_id` VARCHAR(32) NOT NULL,
    `product_name` VARCHAR(32),
    `event_count` VARCHAR(32),
    PRIMARY KEY (`range_time`,`product_id`)
)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4;
```

Step 4: Create an Enhanced Datasource Connection

- **Connecting DLI to Kafka**

  a. On the Kafka management console, click an instance name on the **DMS for Kafka** page. Basic information of the Kafka instance is displayed.

  b. In the **Connection** pane, obtain the **Instance Address (Private Network)**. In the **Network** pane, obtain the VPC and subnet of the instance.

  c. Click the security group name in the **Network** pane. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

  d. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

  e. In the displayed dialog box, set the following parameters:

     - **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_kafka**.
     - **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**. (Queues that are not added to a resource pool are displayed in this list.)
     - **VPC**: Select the VPC of the Kafka instance.

- **Subnet**: Select the subnet of Kafka instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

f. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.

g. In the displayed dialog box, enter *Kafka instance address (private network)*:*port* in the **Address** box and click **Test** to check whether the instance is reachable.

- **Connecting DLI to RDS**

a. Go to the RDS console, click the name of the target RDS DB instance on the **Instances** page. Basic information of the instance is displayed.

b. In the **Connection Information** pane, obtain the floating IP address, database port, VPC, and subnet.

c. Click the security group name. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

d. Check whether the Kafka instance and RDS DB instance are in the same VPC and subnet.

i. If they are, go to **g**. You do not need to create an enhanced datasource connection again.

ii. If they are not, go to **e**. Create an enhanced datasource connection to connect DLI to the subnet where the RDS DB instance locates.

e. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

f. In the displayed dialog box, set the following parameters:

- **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_rds**.
- **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**. (Queues that are not added to a resource pool are displayed in this list.)
- **VPC**: Select the VPC of the RDS DB instance.
- **Subnet**: Select the subnet of RDS DB instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

g. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.

h. In the displayed dialog box, enter *floating IP address*:*database port* of the RDS DB instance you have obtained in **b** in the **Address** box and click **Test** to check whether the database is reachable.

Step 5: Run a Job

1. On the DLI management console, choose **Job Management** > **Flink Jobs**. On the **Flink Jobs** page, click **Create Job**.
2. In the **Create Job** dialog box, set **Type** to **Flink OpenSource SQL** and **Name** to **FlinkKafkaRds**. Click **OK**.
3. On the job editing page, set the following parameters and retain the default values of other parameters.
   - **Queue**: Select the queue created in **Step 1: Create a Queue**.
   - **Flink Version**: Select **1.12**.
   - **Save Job Log**: Enable this function.
   - **OBS Bucket**: Select an OBS bucket for storing job logs and grant access permissions of the OBS bucket as prompted.
   - **Enable Checkpointing**: Enable this function.
   - Enter a SQL statement in the editing pane. The following is an example. Modify the parameters in bold as you need.

     NOTE

     In this example, the syntax version of Flink OpenSource SQL is 1.12. In this example, the data source is Kafka and the result data is written to RDS.

     For details, see **Kafka Source Table** and **JDBC Result Table** (RDS connection).

```
create table click_product(
  user_id string, --ID of the user
  user_name string, --Username
  event_time string, --Click time
  product_id string, --Offering ID
  product_name string --Offering name
) with (
  "connector" = "kafka",
  "properties.bootstrap.servers" = " 10.128.0.120:9092,10.128.0.89:9092,10.128.0.83:9092 ",--
Internal network address and port number of the Kafka instance
  "properties.group.id" = "click",
  "topic" = " testkafkatopic ",--Name of the created Kafka topic
  "format" = "json",
  "scan.startup.mode" = "latest-offset"
);

--Result table
create table top_product (
  range_time string, --Calculated time range
  product_id string, --Offering ID
  product_name string --Offering name
  event_count bigint, --Number of clicks
  primary key (range_time, product_id) not enforced
) with (
  "connector" = "jdbc",
  "url" = "jdbc:mysql://192.168.12.148:3306/testrdsdb ",--testrdsdb indicates the name of the
created RDS database. Replace the IP address and port number with those of the RDS DB
instance.
  "table-name" = "clicktop",
```



```
    "pwd_auth_name"="xxxxx", -- Name of the datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for the job.
    "sink.buffer-flush.max-rows" = "1000",
    "sink.buffer-flush.interval" = "1s"
);

create view current_event_view
as
    select product_id, product_name, count(1) as click_count, concat(substring(event_time, 1, 13), ":00:00") as min_event_time, concat(substring(event_time, 1, 13), ":59:59") as max_event_time
    from click_product group by substring (event_time, 1, 13), product_id, product_name;

insert into top_product
    select
        concat(min_event_time, " - ", max_event_time) as range_time,
        product_id,
        product_name,
        click_count
    from (
        select *,
        row_number() over (partition by min_event_time order by click_count desc) as row_num
        from current_event_view
    )
    where row_num <= 3
```

4. Click **Check Semantic** and ensure that the SQL statement passes the check. Click **Save**. Click **Start**, confirm the job parameters, and click **Start Now** to execute the job. Wait until the job status changes to **Running**.

Step 6: Send Data and Query Results

1. Use the Kafka client to send data to topics created in **Step 2: Create a Kafka Topic** to simulate real-time data streams.

   For details about how Kafka creates and retrieves data, visit **Connecting to a DMS for Kafka Instance**.

   The sample data is as follows:

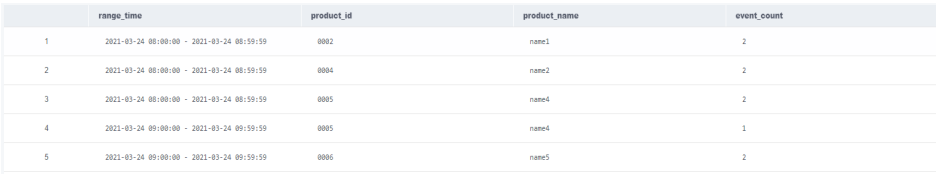
```
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:01:00", "product_id":"0002", "product_name":"name1"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:02:00", "product_id":"0002", "product_name":"name1"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:06:00", "product_id":"0004", "product_name":"name2"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:10:00", "product_id":"0003", "product_name":"name3"}
{"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:15:00", "product_id":"0005", "product_name":"name4"}
{"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:16:00", "product_id":"0005", "product_name":"name4"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:56:00", "product_id":"0004", "product_name":"name2"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:05:00", "product_id":"0005", "product_name":"name4"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:10:00", "product_id":"0006", "product_name":"name5"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 09:13:00", "product_id":"0006", "product_name":"name5"}
```

2. Log in to the RDS console, click the name of the RDS DB instance. On the displayed page, click the name of the created database, for example, **testrdsdb**, and click **Query SQL Statements** in the **Operation** column of the row that containing the **clicktop** table.

```
select * from `clicktop`;
```


3. On the displayed page, click **Execute SQL**. Check whether data has been written into the RDS table.

Figure 3-2 RDS table data



	range_time	product_id	product_name	event_count
1	2021-03-24 08:00:00 - 2021-03-24 08:59:59	0002	name1	2
2	2021-03-24 08:00:00 - 2021-03-24 08:59:59	0004	name2	2
3	2021-03-24 08:00:00 - 2021-03-24 08:59:59	0005	name4	2
4	2021-03-24 09:00:00 - 2021-03-24 09:59:59	0005	name4	1
5	2021-03-24 09:00:00 - 2021-03-24 09:59:59	0006	name5	2

3.2.2 Reading Data from Kafka and Writing Data to GaussDB(DWS)

NOTICE

This guide provides reference for Flink 1.12 only.

Description

This example analyzes real-time vehicle driving data and collects statistics on data results that meet specific conditions. The real-time vehicle driving data is stored in the Kafka source table, and then the analysis result is output to GaussDB(DWS).

For example, enter the following sample data:

```
{"car_id":"3027", "car_owner":"lilei", "car_age":"7", "average_speed":"76", "total_miles":"15000"}
{"car_id":"3028", "car_owner":"hanmeimei", "car_age":"6", "average_speed":"92", "total_miles":"17000"}
{"car_id":"3029", "car_owner":"Ann", "car_age":"10", "average_speed":"81", "total_miles":"230000"}
```

Expected output is vehicles meeting the average_speed <= 90 and total_miles <= 200,000 condition.

```
{"car_id":"3027", "car_owner":"lilei", "car_age":"7", "average_speed":"76", "total_miles":"15000"}
```

Prerequisites

1. You have created a DMS for Kafka instance.

   For details, see **Getting Started with DMS for Kafka**.

   CAUTION

   When you create the instance, do not enable **Kafka SASL_SSL**.

2. You have created a GaussDB(DWS) instance.

   For details about how to create a GaussDB(DWS) cluster, see **Creating a Cluster**.

Overall Development Process

Overall Process

Figure 3-3 Job development process

Step 1: Create a Queue

Step 2: Create a Kafka Topic

Step 3: Create a GaussDB(DWS) Database and Table

Step 4: Create an Enhanced Datasource Connection

Step 5: Run a Job

Step 6: Send Data and Query Results

Step 1: Create a Queue

1. Log in to the DLI console. In the navigation pane on the left, choose **Resources** > **Queue Management**.
2. On the displayed page, click **Buy Queue** in the upper right corner.
3. On the **Buy Queue** page, set queue parameters as follows:
   - **Billing Mode**: .
   - **Region** and **Project**: Retain the default values.
   - **Name**: Enter a queue name.

     NOTE

     The queue name can contain only digits, letters, and underscores (_), but cannot contain only digits or start with an underscore (_). The name must contain 1 to 128 characters.

     The queue name is case-insensitive. Uppercase letters will be automatically converted to lowercase letters.
   - **Type**: Select **For general purpose**. Select the **Dedicated Resource Mode**.
   - **AZ Mode** and **Specifications**: Retain the default values.
   - **Enterprise Project**: Select **default**.
   - **Advanced Settings**: Select **Custom**.
   - **CIDR Block**: Specify the queue network segment. For example, **10.0.0.0/16**.

     CAUTION

     The CIDR block of a queue cannot overlap with the CIDR blocks of DMS Kafka and RDS for MySQL DB instances. Otherwise, datasource connections will fail to be created.
   - Set other parameters as required.
4. Click **Buy**. Confirm the configuration and click **Submit**.

Step 2: Create a Kafka Topic

1. On the Kafka management console, click an instance name on the **DMS for Kafka** page. Basic information of the Kafka instance is displayed.
2. Choose **Topics** in the navigation pane on the left. On the displayed page, click **Create Topic**. Configure the following parameters:
   - **Topic Name**: For this example, enter **testkafkatopic**.
   - **Partitions**: Set the value to **1**.
   - **Replicas**: Set the value to **1**.

   Retain default values for other parameters.

Step 3: Create a GaussDB(DWS) Database and Table

1. Connect to the created GaussDB(DWS) cluster by referring to **Using the gsql CLI Client to Connect to a Cluster**.
2. Connect to the default database **gaussdb** of a GaussDB(DWS) cluster.
   ```
   gsql -d gaussdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```
   - **gaussdb**: Default database of the GaussDB(DWS) cluster
   - **Connection address of the DWS cluster**: If a public network address is used for connection, set this parameter to the public network IP address or domain name. If a private network address is used for connection, set this parameter to the private network IP address or domain name. For details, see section **Obtaining the Cluster Connection Address**. If an ELB is used for connection, set this parameter to the ELB address.
   - **dbadmin**: Default administrator username used during cluster creation
   - **password**: Default password of the administrator
3. Run the following command to create the **testdwsdb** database:
   ```
   CREATE DATABASE testdwsdb;
   ```
4. Run the following command to exit the **gaussdb** database and connect to **testdwsdb**:
   ```
   \q
   gsql -d testdwsdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```
5. Run the following commands to create a table:
   ```
   create schema test;
   set current_schema= test;
   drop table if exists qualified_cars;
   CREATE TABLE qualified_cars
   (
       car_id VARCHAR,
       car_owner VARCHAR,
       car_age INTEGER ,
       average_speed FLOAT8,
       total_miles FLOAT8
   );
   ```

Step 4: Create an Enhanced Datasource Connection

- **Connecting DLI to Kafka**
  a. On the Kafka management console, click an instance name on the **DMS for Kafka** page. Basic information of the Kafka instance is displayed.

b. In the **Connection** pane, obtain the **Instance Address (Private Network)**. In the **Network** pane, obtain the VPC and subnet of the instance.

c. Click the security group name in the **Network** pane. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

d. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

e. In the displayed dialog box, set the following parameters: For details, see the following section:

- **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_kafka**.
- **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.
- **VPC**: Select the VPC of the Kafka instance.
- **Subnet**: Select the subnet of Kafka instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

f. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.

g. In the displayed dialog box, enter *Kafka instance address (private network)*:*port* in the **Address** box and click **Test** to check whether the instance is reachable.

- **Connecting DLI to GaussDB(DWS)**

a. On the GaussDB(DWS) management console, choose **Clusters**. On the displayed page, click the name of the created GaussDB(DWS) cluster to view basic information.

b. In the Basic Information tab, locate the **Database Attributes** pane and obtain the private IP address and port number of the DB instance. In the **Network** pane, obtain VPC, and subnet information.

c. Click the security group name. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

d. Check whether the Kafka instance and GaussDB(DWS) instance are in the same VPC and subnet.

i. If they are, go to **g**. You do not need to create an enhanced datasource connection again.

ii. If they are not, go to **e**. Create an enhanced datasource connection to connect DLI to the subnet where the GaussDB(DWS) instance locates.

e. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

f. In the displayed dialog box, set the following parameters: For details, see the following section:

- **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_dws**.
- **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.
- **VPC**: Select the VPC of the GaussDB(DWS) instance.
- **Subnet**: Select the subnet of GaussDB(DWS) instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

g. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.

h. In the displayed dialog box, enter *floating IP address*:*database port* of the GaussDB(DWS) instance you have obtained in **b** in the **Address** box and click **Test** to check whether the database is reachable.

Step 5: Run a Job

1. On the DLI management console, choose **Job Management** > **Flink Jobs**. On the **Flink Jobs** page, click **Create Job**.
2. In the **Create Job** dialog box, set **Type** to **Flink OpenSource SQL** and **Name** to **FlinkKafkaDWS**. Click **OK**.
3. On the job editing page, set the following parameters and retain the default values of other parameters.
   - **Queue**: Select the queue created in **Step 1: Create a Queue**.
   - **Flink Version**: Select **1.12**.
   - **Save Job Log**: Enable this function.
   - **OBS Bucket**: Select an OBS bucket for storing job logs and grant access permissions of the OBS bucket as prompted.
   - **Enable Checkpointing**: Enable this function.
   - Enter a SQL statement in the editing pane. The following is an example. Modify the parameters in bold as you need.

NOTE

In this example, the syntax version of Flink OpenSource SQL is 1.12. In this example, the data source is Kafka and the result data is written to GaussDB(DWS).

For details, see **Kafka Source Table** and **GaussDB(DWS) Result Table**.

```
create table car_infos(
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed DOUBLE,
  total_miles DOUBLE
) with (
    "connector" = "kafka",
    "properties.bootstrap.servers" = " 10.128.0.120:9092,10.128.0.89:9092,10.128.0.83:9092 ",--Internal network address and port number of the Kafka instance
    "properties.group.id" = "click",
    "topic" = " testkafkatopic",--Created Kafka topic
    "format" = "json",
    "scan.startup.mode" = "latest-offset"
);

create table qualified_cars (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed DOUBLE,
  total_miles DOUBLE
)
WITH (
  'connector' = 'gaussdb',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url'='jdbc:gaussdb://192.168.168.16:8000/testdwsdb ', ---192.168.168.16:8000 indicates the internal IP address and port of the GaussDB(DWS) instance. testdwsdb indicates the name of the created GaussDB(DWS) database.
  'table-name' = ' test\".\"qualified_cars', ---test indicates the schema of the created GaussDB(DWS) table, and qualified_cars indicates the GaussDB(DWS) table name.
  'pwd_auth_name'= 'xxxxx', -- Name of the datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for the job.
  'write.mode' = 'insert'
);

/** Output information about qualified vehicles **/
INSERT INTO qualified_cars
SELECT *
FROM car_infos
where average_speed <= 90 and total_miles <= 200000;
```

4. Click **Check Semantic** and ensure that the SQL statement passes the check. Click **Save**. Click **Start**, confirm the job parameters, and click **Start Now** to execute the job. Wait until the job status changes to **Running**.

Step 6: Send Data and Query Results

1. Use the Kafka client to send data to topics created in **Step 2: Create a Kafka Topic** to simulate real-time data streams.

   For details about how Kafka creates and retrieves data, visit **Connecting to a DMS for Kafka Instance**.

   The sample data is as follows:

```
{"car_id":"3027", "car_owner":"lilei", "car_age":"7", "average_speed":"76", "total_miles":"15000"}
{"car_id":"3028", "car_owner":"hanmeimei", "car_age":"6", "average_speed":"92", "total_miles":"17000"}
{"car_id":"3029", "car_owner":"Ann", "car_age":"10", "average_speed":"81", "total_miles":"230000"}
```


2. Connect to the created GaussDB(DWS) cluster.

   For details, see **Using the gsql CLI Client to Connect to a Cluster**.

3. Connect to the default database **testdwsdb** of a GaussDB(DWS) cluster.

   ```
   gsql -d testdwsdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```

4. Run the following statement to query GaussDB(DWS) table data:

   ```
   select * from test.qualified_cars;
   ```

   The query result is as follows:

   ```
   car_id  car_owner  car_age  average_speed  total_miles
   3027      lilei     7          76.0        15000.0
   ```

3.2.3 Reading Data from Kafka and Writing Data to Elasticsearch

NOTICE

This guide provides reference for Flink 1.12 only.

Description

This example analyzes offering purchase data and collects statistics on data results that meet specific conditions. The offering purchase data is stored in the Kafka source table, and then the analysis result is output to Elasticsearch .

For example, enter the following sample data:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0002", "user_name":"Jason", "area_id":"330106"}
```

DLI reads data from Kafka and writes the data to Elasticsearch. You can view the result in Kibana of the Elasticsearch cluster.

Prerequisites

1. You have created a DMS for Kafka instance.

   For details, see **Getting Started with DMS for Kafka**.

   CAUTION

   When you create a DMS Kafka instance, do not enable **Kafka SASL_SSL**.

2. You have created a CSS Elasticsearch cluster.

   For how to create a CSS cluster, see **Creating a CSS Cluster**.

   In this example, the version of the created CSS cluster is 7.6.2, and security mode is disabled for the cluster.

Overall Process

Figure 3-4 shows the overall development process.

Figure 3-4 Job development process

Step 1: Create a Queue

Step 2: Create a Kafka Topic

Step 3: Create an Elasticsearch Index

Step 4: Create an Enhanced Datasource Connection

Step 5: Run a Job

Step 6: Send Data and Query Results

Step 1: Create a Queue

1. Log in to the DLI console. In the navigation pane on the left, choose **Resources** > **Queue Management**.
2. On the displayed page, click **Buy Queue** in the upper right corner.
3. On the **Buy Queue** page, set queue parameters as follows:
   - **Billing Mode**: .
   - **Region** and **Project**: Retain the default values.
   - **Name**: Enter a queue name.

     NOTE

     The queue name can contain only digits, letters, and underscores (_), but cannot contain only digits or start with an underscore (_). The name must contain 1 to 128 characters.

     The queue name is case-insensitive. Uppercase letters will be automatically converted to lowercase letters.
   - **Type**: Select **For general purpose**. Select the **Dedicated Resource Mode**.
   - **AZ Mode** and **Specifications**: Retain the default values.
   - **Enterprise Project**: Select **default**.
   - **Advanced Settings**: Select **Custom**.
   - **CIDR Block**: Specify the queue network segment. For example, **10.0.0.0/16**.

     CAUTION

     The CIDR block of a queue cannot overlap with the CIDR blocks of DMS Kafka and RDS for MySQL DB instances. Otherwise, datasource connections will fail to be created.

- Set other parameters as required.

4. Click **Buy**. Confirm the configuration and click **Submit**.

Step 2: Create a Kafka Topic

1. On the Kafka management console, click an instance name on the **DMS for Kafka** page. Basic information of the Kafka instance is displayed.
2. Choose **Topics** in the navigation pane on the left. On the displayed page, click **Create Topic**. Configure the following parameters:
   - **Topic Name**: For this example, enter **testkafkatopic**.
   - **Partitions**: Set the value to **1**.
   - **Replicas**: Set the value to **1**.

   Retain default values for other parameters.

Step 3: Create an Elasticsearch Index

1. Log in to the CSS management console and choose **Clusters** > **Elasticsearch** from the navigation pane on the left.
2. On the **Clusters** page, click **Access Kibana** in the **Operation** column of the created CSS cluster.
3. On the displayed page, choose **Dev Tools** in the navigation pane on the left. The **Console** page is displayed.
4. On the displayed page, run the following command to create index **shoporders**:

```
PUT /shoporders
{
  "settings": {
    "number_of_shards": 1
  },
    "mappings": {
      "properties": {
        "order_id": {
          "type": "text"
        },
        "order_channel": {
          "type": "text"
        },
        "order_time": {
          "type": "text"
        },
        "pay_amount": {
          "type": "double"
        },
        "real_pay": {
          "type": "double"
        },
        "pay_time": {
          "type": "text"
        },
        "user_id": {
          "type": "text"
        },
        "user_name": {
          "type": "text"
        },
        "area_id": {
          "type": "text"
        }
      }
```



```
    }
}
```

Step 4: Create an Enhanced Datasource Connection

- **Connecting DLI to Kafka**
  a. On the Kafka management console, click an instance name on the **DMS for Kafka** page. Basic information of the Kafka instance is displayed.
  b. In the **Connection** pane, obtain the **Instance Address (Private Network)**. In the **Network** pane, obtain the VPC and subnet of the instance.
  c. Click the security group name in the **Network** pane. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.
  d. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.
  e. In the displayed dialog box, set the following parameters: For details, see the following section:
     - **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_kafka**.
     - **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.
     - **VPC**: Select the VPC of the Kafka instance.
     - **Subnet**: Select the subnet of Kafka instance.
     - Set other parameters as you need.

     Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.
  f. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.
  g. In the displayed dialog box, enter *Kafka instance address (private network)*:*port* in the **Address** box and click **Test** to check whether the instance is reachable.
- **Connecting DLI to CSS**
  a. On the CSS management console, choose **Clusters** > **Elasticsearch**. On the displayed page, click the name of the created CSS cluster to view basic information.
  b. On the **Cluster Information** page, obtain the **Private Network Address**, **VPC**, AND **Subnet**.
  c. Click the security group name. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to

Allow, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

d. Check whether the Kafka instance and Elasticsearch instance are in the same VPC and subnet.

   i. If they are, go to **g**. You do not need to create an enhanced datasource connection again.

   ii. If they are not, go to **e**. Create an enhanced datasource connection to connect DLI to the subnet where the Elasticsearch instance locates.

e. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

f. In the displayed dialog box, set the following parameters: For details, see the following section:

   - **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_css**.
   - **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.
   - **VPC**: Select the VPC of the Elasticsearch instance.
   - **Subnet**: Select the subnet of Elasticsearch instance.
   - Set other parameters as you need.

   Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

g. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.

h. In the displayed dialog box, enter *floating IP address*:*database port* of the Elasticsearch instance you have obtained in **b** in the **Address** box and click **Test** to check whether the database is reachable.

Step 5: Run a Job

1. On the DLI management console, choose **Job Management** > **Flink Jobs**. On the **Flink Jobs** page, click **Create Job**.
2. In the the **Create Job** dialog box, set **Type** to **Flink OpenSource SQL** and **Name** to **FlinkKafkaES**. Click **OK**.
3. On the job editing page, set the following parameters and retain the default values of other parameters.
   - **Queue**: Select the queue created in **Step 1: Create a Queue**.
   - **Flink Version**: Select **1.12**.
   - **Save Job Log**: Enable this function.
   - **OBS Bucket**: Select an OBS bucket for storing job logs and grant access permissions of the OBS bucket as prompted.

- **Enable Checkpointing**: Enable this function.
- Enter a SQL statement in the editing pane. The following is an example. Modify the parameters in bold as you need.

  NOTE

  In this example, the syntax version of Flink OpenSource SQL is 1.12. In this example, the data source is Kafka and the result data is written to Elasticsearch.

  For details, see **Kafka Source Table** and **Elasticsearch Result Table**.

- Create a Kafka source table and connect DLI to the Kafka data source.

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) with (
  "connector" = "kafka",
  "properties.bootstrap.servers" = "10.128.0.120:9092,10.128.0.89:9092,10.128.0.83:9092",--
Internal network address and port number of the Kafka instance
  "properties.group.id" = "click",
  "topic" = "testkafkatopic",--Created Kafka topic
  "format" = "json",
  "scan.startup.mode" = "latest-offset"
);
```

- Create an Elasticsearch result table to display the data analyzed by DLI.

```
CREATE TABLE elasticsearchSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) WITH (
  'connector' = 'elasticsearch-7',
  'hosts' = '192.168.168.125:9200', --Private IP address and port of the CSS cluster
  'index' = 'shoporders' --Created Elasticsearch engine
);
--Write Kafka data to Elasticsearch indexes
insert into
  elasticsearchSink
select
  *
from
  kafkaSource;
```

4. Click **Check Semantic** and ensure that the SQL statement passes the check. Click **Save**. Click **Start**, confirm the job parameters, and click **Start Now** to execute the job. Wait until the job status changes to **Running**.

Step 6: Send Data and Query Results

1. Kafaka sends data.

   Use the Kafka client to send data to topics created in **Step 2: Create a Kafka Topic** to simulate real-time data streams.

   For details about how Kafka creates and retrieves data, visit **Connecting to a DMS for Kafka Instance**.

The sample data is as follows:

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0002", "user_name":"Jason", "area_id":"330106"}
```

2. View the data processing result on Elasticsearch.

   After the message is sent to Kafka, run the following statement in Kibana for the CSS cluster and check the result:

   ```
   GET shoporders/_search
   ```

   The query result is as follows:

   ```
   {
     "took" : 0,
     "timed_out" : false,
     "_shards" : {
       "total" : 1,
       "successful" : 1,
       "skipped" : 0,
       "failed" : 0
     },
     "hits" : {
       "total" : {
         "value" : 2,
         "relation" : "eq"
       },
       "max_score" : 1.0,
       "hits" : [
         {
           "_index" : "shoporders",
           "_type" : "_doc",
           "_id" : "6fswzIAByVjqg3_qAyM1",
           "_score" : 1.0,
           "_source" : {
             "order_id" : "202103241000000001",
             "order_channel" : "webShop",
             "order_time" : "2021-03-24 10:00:00",
             "pay_amount" : 100.0,
             "real_pay" : 100.0,
             "pay_time" : "2021-03-24 10:02:03",
             "user_id" : "0001",
             "user_name" : "Alice",
             "area_id" : "330106"
           }
         },
         {
           "_index" : "shoporders",
           "_type" : "_doc",
           "_id" : "6vs1zIAByVjqg3_qyyPp",
           "_score" : 1.0,
           "_source" : {
             "order_id" : "202103241606060001",
             "order_channel" : "appShop",
             "order_time" : "2021-03-24 16:06:06",
             "pay_amount" : 200.0,
             "real_pay" : 180.0,
             "pay_time" : "2021-03-24 16:10:06",
             "user_id" : "0002",
             "user_name" : "Jason",
             "area_id" : "330106"
           }
         }
       ]
     }
   }
   ```


3.2.4 Reading Data from MySQL CDC and Writing Data to GaussDB(DWS)

NOTICE

This guide provides reference for Flink 1.12 only.

Description

Change Data Capture (CDC) can synchronize incremental changes from the source database to one or more destinations. During data synchronization, CDC processes data, for example, grouping (GROUP BY) and joining multiple tables (JOIN).

This example creates a MySQL CDC source table to monitor MySQL data changes and insert the changed data into a GaussDB(DWS) database.

Prerequisites

1. You have created an RDS for MySQL DB instance. In this example, the RDS for MySQL database version is 8.0.

   For details, see **Buying an RDS for MySQL DB Instance**.
2. You have created a GaussDB(DWS) instance.

   For how to create a GaussDB(DWS) cluster, see **Creating a GaussDB(DWS) Cluster**.

Overall Process

Figure 3-5 shows the overall development process.

Figure 3-5 Job development process

Step 1: Create a Queue

Step 2: Create an RDS MySQL Database and Table

Step 3: Create a GaussDB(DWS) Database and Table

Step 4: Create an Enhanced Datasource Connection

Step 5: Run a Job

Step 6: Send Data and Query Results

Step 1: Create a Queue

1. Log in to the DLI console. In the navigation pane on the left, choose **Resources** > **Queue Management**.
2. On the displayed page, click **Buy Queue** in the upper right corner.

3. On the **Buy Queue** page, set queue parameters as follows:
   - **Billing Mode**: .
   - **Region** and **Project**: Retain the default values.
   - **Name**: Enter a queue name.

     NOTE

     The queue name can contain only digits, letters, and underscores (_), but cannot contain only digits or start with an underscore (_). The name must contain 1 to 128 characters.

     The queue name is case-insensitive. Uppercase letters will be automatically converted to lowercase letters.
   - **Type**: Select **For general purpose**. Select the **Dedicated Resource Mode**.
   - **AZ Mode** and **Specifications**: Retain the default values.
   - **Enterprise Project**: Select **default**.
   - **Advanced Settings**: Select **Custom**.
   - **CIDR Block**: Specify the queue network segment. For example, **10.0.0.0/16**.

     CAUTION

     The CIDR block of a queue cannot overlap with the CIDR blocks of DMS Kafka and RDS for MySQL DB instances. Otherwise, datasource connections will fail to be created.
   - Set other parameters as required.
4. Click **Buy**. Confirm the configuration and click **Submit**.

Step 2: Create an RDS MySQL Database and Table

1. Log in to the RDS console. On the displayed page, locate the target MySQL DB instance and choose **More** > **Log In** in the **Operation** column.
2. On the displayed login dialog box, enter the username and password and click **Log In**.
3. On the **Databases** page, click **Create Database**. In the displayed dialog box, enter **testrdsdb** as the database name and retain default values of rest parameters. Then, click **OK**.
4. In the **Operation** column of row where the created database locates, click **SQL Window** and enter the following statement to create a table:

```
CREATE TABLE mysqlcdc (
    `order_id` VARCHAR(64) NOT NULL,
    `order_channel` VARCHAR(32) NOT NULL,
    `order_time` VARCHAR(32),
    `pay_amount` DOUBLE,
    `real_pay` DOUBLE,
    `pay_time` VARCHAR(32),
    `user_id` VARCHAR(32),
    `user_name` VARCHAR(32),
    `area_id` VARCHAR(32)

)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4;
```


Step 3: Create a GaussDB(DWS) Database and Table

1. Connect to the created GaussDB(DWS) cluster.

   For details, see **Using the gsql CLI Client to Connect to a Cluster**.

2. Connect to the default database **gaussdb** of a GaussDB(DWS) cluster.

   ```
   gsql -d gaussdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```

   - **gaussdb**: Default database of the GaussDB(DWS) cluster
   - **Connection address of the DWS cluster**: If a public network address is used for connection, set this parameter to the public network IP address or domain name. If a private network address is used for connection, set this parameter to the private network IP address or domain name. For details, see section **Obtaining the Cluster Connection Address**. If an ELB is used for connection, set this parameter to the ELB address.
   - **dbadmin**: Default administrator username used during cluster creation
   - **-W**: Default password of the administrator

3. Run the following command to create the **testdwsdb** database:

   ```
   CREATE DATABASE testdwsdb;
   ```

4. Run the following command to exit the **gaussdb** database and connect to **testdwsdb**:

   ```
   \q
   gsql -d testdwsdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```

5. Run the following commands to create a table:

   ```
   create schema test;
   set current_schema= test;
   drop table if exists dwsresult;
   CREATE TABLE dwsresult
   (
       car_id VARCHAR,
       car_owner VARCHAR,
       car_age INTEGER ,
       average_speed FLOAT8,
       total_miles FLOAT8
   );
   ```

Step 4: Create an Enhanced Datasource Connection

- **Connecting DLI to RDS**

  a. Go to the RDS console, click the name of the target RDS DB instance on the **Instances** page. Basic information of the instance is displayed.

  b. In the **Connection Information** pane, obtain the floating IP address, database port, VPC, and subnet.

  c. Click the security group name. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

  d. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

  e. In the displayed dialog box, set the following parameters: For details, see the following section:

- **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_rds**.
- **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.
- **VPC**: Select the VPC of the RDS DB instance.
- **Subnet**: Select the subnet of RDS DB instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

f. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.

g. In the displayed dialog box, enter *floating IP address*:*database port* of the RDS DB instance you have obtained in **b** in the **Address** box and click **Test** to check whether the database is reachable.

- **Connecting DLI to GaussDB(DWS)**

a. On the GaussDB(DWS) management console, choose **Clusters**. On the displayed page, click the name of the created GaussDB(DWS) cluster to view basic information.

b. In the Basic Information tab, locate the **Database Attributes** pane and obtain the private IP address and port number of the DB instance. In the **Network** pane, obtain VPC, and subnet information.

c. Click the security group name. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

d. Check whether the RDS instance and GaussDB(DWS) instance are in the same VPC and subnet.

i. If they are, go to **g**. You do not need to create an enhanced datasource connection again.

ii. If they are not, go to **e**. Create an enhanced datasource connection to connect RDS to the subnet where the GaussDB(DWS) instance locates.

e. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

f. In the displayed dialog box, set the following parameters: For details, see the following section:

- **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_dws**.
- **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.

- **VPC**: Select the VPC of the GaussDB(DWS) instance.
- **Subnet**: Select the subnet of GaussDB(DWS) instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

g. Choose **Resources** > **Queue Management** from the navigation pane, locate the queue you created in **Step 1: Create a Queue**. In the **Operation** column, click **More** > **Test Address Connectivity**.

h. In the displayed dialog box, enter *floating IP address*:*database port* of the GaussDB(DWS) instance you have obtained in **b** in the **Address** box and click **Test** to check whether the database is reachable.

Step 5: Run a Job

1. On the DLI management console, choose **Job Management** > **Flink Jobs**. On the **Flink Jobs** page, click **Create Job**.
2. In the **Create Job** dialog box, set **Type** to **Flink OpenSource SQL** and **Name** to **FlinkCDCMySQLDWS**. Click **OK**.
3. On the job editing page, set the following parameters and retain the default values of other parameters.
   - **Queue**: Select the queue created in **Step 1: Create a Queue**.
   - **Flink Version**: Select **1.12**.
   - **Save Job Log**: Enable this function.
   - **OBS Bucket**: Select an OBS bucket for storing job logs and grant access permissions of the OBS bucket as prompted.
   - **Enable Checkpointing**: Enable this function.
   - Enter a SQL statement in the editing pane. The following is an example. Modify the parameters in bold as you need.

     NOTE

     In this example, the syntax version of Flink OpenSource SQL is 1.12. In this example, the data source is Kafka and the result data is written to Elasticsearch.

     For details, see **MySQL CDC Source Table** and **GaussDB(DWS) Result Table**.

```
create table mysqlCdcSource(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING
) with (
  'connector' = 'mysql-cdc',
  'hostname' = ' 192.168.12.148',--IP address of the RDS MySQL instance
  'port'= ' 3306',--Port number of the RDS MySQL instance
  'pwd_auth_name'= 'xxxxx', -- Name of the datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for the job.
```



```
  'database-name' = ' testrdsdb',--Database name of the RDS MySQL instance
  'table-name' = ' mysqlcdc'--Name of the target table in the database
);

create table dwsSink(
  order_channel string,
  pay_amount double,
  real_pay double,
  primary key(order_channel) not enforced
) with (
  'connector' = 'gaussdb',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url'='jdbc:gaussdb://192.168.168.16:8000/testdwsdb ', ---192.168.168.16:8000 indicates the
internal IP address and port of the GaussDB(DWS) instance. testdwsdb indicates the name of
the created GaussDB(DWS) database.
  'table-name' = ' test\".\"dwsresult', ---test indicates the schema of the created
GaussDB(DWS) table, and dwsresult indicates the GaussDB(DWS) table name.
  'pwd_auth_name'= 'xxxxx', -- Name of the datasource authentication of the password type
created on DLI. If datasource authentication is used, you do not need to set the username and
password for the job.
  'write.mode' = 'insert'
);

insert into dwsSink select order_channel, sum(pay_amount),sum(real_pay) from
mysqlCdcSource group by order_channel;
```

4. Click **Check Semantic** and ensure that the SQL statement passes the check. Click **Save**. Click **Start**, confirm the job parameters, and click **Start Now** to execute the job. Wait until the job status changes to **Running**.

Step 6: Send Data and Query Results

1. Log in to the RDS console. On the displayed page, locate the target MySQL DB instance and choose **More** > **Log In** in the **Operation** column.
2. On the displayed login dialog box, enter the username and password and click **Log In**.
3. In the **Operation** column of row where the created database locates, click **SQL Window** and enter the following statement to create a table and insert data to the table:

   ```
   insert into mysqlcdc values
   ('202103241000000001','webShop','2021-03-24 10:00:00','100.00','100.00','2021-03-24
   10:02:03','0001','Alice','330106'),
   ('202103241206060001','appShop','2021-03-24 12:06:06','200.00','180.00','2021-03-24
   16:10:06','0002','Jason','330106'),
   ('202103241403000001','webShop','2021-03-24 14:03:00','300.00','100.00','2021-03-24
   10:02:03','0003','Lily','330106'),
   ('202103241636060001','appShop','2021-03-24 16:36:06','200.00','150.00','2021-03-24
   16:10:06','0001','Henry','330106');
   ```

4. Connect to the created GaussDB(DWS) cluster by referring to **Using the gsql CLI Client to Connect to a Cluster**.
5. Connect to the default database **testdwsdb** of a GaussDB(DWS) cluster.

   ```
   gsql -d testdwsdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W
   password -r
   ```

6. Run the following statements to query table data:

   ```
   select * from test.dwsresult;
   ```

   The query result is as follows:

   ```
   order_channel  pay_amount  real_pay
   appShop         400.0       330.0
   webShop         400.0       200.0
   ```


3.2.5 Reading Data from PostgreSQL CDC and Writing Data to GaussDB(DWS)

NOTICE

This guide provides reference for Flink 1.12 only.

Description

Change Data Capture (CDC) can synchronize incremental changes from the source database to one or more destinations. During data synchronization, CDC processes data, for example, grouping (GROUP BY) and joining multiple tables (JOIN).

This example creates a PostgreSQL CDC source table to monitor PostgreSQL data changes and insert the changed data into a GaussDB(DWS) database.

Prerequisites

1. You have created an RDS for PostgreSQL DB instance. In this example, the RDS for PostgreSQL database version is 11.

   For details, see **Getting Started with RDS for PostgreSQL**.

   NOTE

   The version of the RDS PostgreSQL database cannot be earlier than 11.

2. You have created a GaussDB(DWS) instance.

   For details about how to create a GaussDB(DWS) cluster, see **Creating a Cluster**.

Overall Process

Figure 3-6 shows the overall development process.

Figure 3-6 Job development process

Step 1: Create a Queue

Step 2: Create an RDS PostgreSQL Database and Table

Step 3: Create a GaussDB(DWS) Database and Table

Step 4: Create an Enhanced Datasource Connection

Step 5: Run a Job

Step 6: Send Data and Query Results

Step 1: Create a Queue

1. Log in to the DLI console. In the navigation pane on the left, choose **Resources** > **Queue Management**.
2. On the displayed page, click **Buy Queue** in the upper right corner.
3. On the **Buy Queue** page, set queue parameters as follows:
   - **Billing Mode**: .
   - **Region** and **Project**: Retain the default values.
   - **Name**: Enter a queue name.

     NOTE

     The queue name can contain only digits, letters, and underscores (_), but cannot contain only digits or start with an underscore (_). The name must contain 1 to 128 characters.

     The queue name is case-insensitive. Uppercase letters will be automatically converted to lowercase letters.
   - **Type**: Select **For general purpose**. Select the **Dedicated Resource Mode**.
   - **AZ Mode** and **Specifications**: Retain the default values.
   - **Enterprise Project**: Select **default**.
   - **Advanced Settings**: Select **Custom**.
   - **CIDR Block**: Specify the queue network segment. For example, **10.0.0.0/16**.

     CAUTION

     The CIDR block of a queue cannot overlap with the CIDR blocks of DMS Kafka and RDS for MySQL DB instances. Otherwise, datasource connections will fail to be created.
   - Set other parameters as required.
4. Click **Buy**. Confirm the configuration and click **Submit**.

Step 2: Create an RDS PostgreSQL Database and Table

1. Log in to the RDS console. On the displayed page, locate the target PostgreSQL DB instance and choose **More** > **Log In** in the **Operation** column.
2. In the login dialog box displayed, enter the username and password and click **Log In**.
3. Create a database instance and name it **testrdsdb**.
4. Create a schema named **test** for the **testrdsdb** database.
5. Choose **SQL Operations** > **SQL Query**. On the page displayed, create an RDS for PostgreSQL table.

```
create table test.cdc_order(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
```



```
  user_name VARCHAR,
  area_id VARCHAR,
  primary key(order_id));
```

Run the following statement in the PostgreSQL instance:

```
ALTER TABLE test.cdc_order REPLICA IDENTITY FULL;
```

Step 3: Create a GaussDB(DWS) Database and Table

1. Connect to the created GaussDB(DWS) cluster.

   For details, see **Using the gsql CLI Client to Connect to a Cluster**.

2. Connect to the default database **gaussdb** of a GaussDB(DWS) cluster.

   ```
   gsql -d gaussdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```

   - **gaussdb**: Default database of the GaussDB(DWS) cluster
   - **Connection address of the DWS cluster**: If a public network address is used for connection, set this parameter to the public network IP address or domain name. If a private network address is used for connection, set this parameter to the private network IP address or domain name. For details, see section **Obtaining the Cluster Connection Address**. If an ELB is used for connection, set this parameter to the ELB address.
   - **dbadmin**: Default administrator username used during cluster creation
   - **-W**: Default password of the administrator

3. Run the following command to create the **testdwsdb** database:

   ```
   CREATE DATABASE testdwsdb;
   ```

4. Run the following command to exit the **gaussdb** database and connect to **testdwsdb**:

   ```
   \q
   gsql -d testdwsdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```

5. Run the following commands to create a table:

   ```
   create schema test;
   set current_schema= test;
   drop table if exists dws_order;
   CREATE TABLE dws_order
   (
     order_id VARCHAR,
     order_channel VARCHAR,
     order_time VARCHAR,
     pay_amount FLOAT8,
     real_pay FLOAT8,
     pay_time VARCHAR,
     user_id VARCHAR,
     user_name VARCHAR,
     area_id VARCHAR
   );
   ```

Step 4: Create an Enhanced Datasource Connection

- **Connecting DLI to RDS**

  a. Go to the RDS console, click the name of the target RDS DB instance on the **Instances** page. Basic information of the instance is displayed.

  b. In the **Connection Information** pane, obtain the floating IP address, database port, VPC, and subnet.

  c. Click the security group name. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if

the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

d. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

e. In the displayed dialog box, set the following parameters: For details, see the following section:

- **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_rds**.
- **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.
- **VPC**: Select the VPC of the RDS DB instance.
- **Subnet**: Select the subnet of RDS DB instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

f. In the navigation pane on the left, choose **Resources** > **Queue Management**. On the page displayed, locate the queue you created in **Step 1: Create a Queue**, click **More** in the **Operation** column, and select **Test Address Connectivity**.

g. In the displayed dialog box, enter *floating IP address*:*database port* of the RDS DB instance you have obtained in **b** in the **Address** box and click **Test** to check whether the database is reachable.

- **Connecting DLI to GaussDB(DWS)**

a. On the GaussDB(DWS) management console, choose **Clusters**. On the displayed page, click the name of the created GaussDB(DWS) cluster to view basic information.

b. In the Basic Information tab, locate the **Database Attributes** pane and obtain the private IP address and port number of the DB instance. In the **Network** pane, obtain VPC, and subnet information.

c. Click the security group name. On the displayed page, click the **Inbound Rules** tab and add a rule to allow access from DLI queues. For example, if the CIDR block of the queue is **10.0.0.0/16**, set **Priority** to **1**, **Action** to **Allow**, **Protocol** to **TCP**, **Type** to **IPv4**, **Source** to **10.0.0.0/16**, and click **OK**.

d. Check whether the RDS instance and GaussDB(DWS) instance are in the same VPC and subnet.

i. If they are, go to **g**. You do not need to create an enhanced datasource connection again.

ii. If they are not, go to **e**. Create an enhanced datasource connection to connect RDS to the subnet where the GaussDB(DWS) instance locates.

e. Log in to the DLI management console. In the navigation pane on the left, choose **Datasource Connections**. On the displayed page, click **Create** in the **Enhanced** tab.

f. In the displayed dialog box, set the following parameters: For details, see the following section:

- **Connection Name**: Enter a name for the enhanced datasource connection. For this example, enter **dli_dws**.
- **Resource Pool**: Select the name of the queue created in **Step 1: Create a Queue**.
- **VPC**: Select the VPC of the GaussDB(DWS) instance.
- **Subnet**: Select the subnet of GaussDB(DWS) instance.
- Set other parameters as you need.

Click **OK**. Click the name of the created datasource connection to view its status. You can perform subsequent steps only after the connection status changes to **Active**.

g. In the navigation pane on the left, choose **Resources** > **Queue Management**. On the page displayed, locate the queue you created in **Step 1: Create a Queue**, click **More** in the **Operation** column, and select **Test Address Connectivity**.

h. In the displayed dialog box, enter *floating IP address*:*database port* of the GaussDB(DWS) instance you have obtained in **b** in the **Address** box and click **Test** to check whether the database is reachable.

Step 5: Run a Job

1. On the DLI management console, choose **Job Management** > **Flink Jobs**. On the **Flink Jobs** page, click **Create Job**.
2. In the **Create Job** dialog box, set **Type** to **Flink OpenSource SQL** and **Name** to **FlinkCDCPostgreDWS**. Click **OK**.
3. On the job editing page, set the following parameters and retain the default values of other parameters.
   - **Queue**: Select the queue created in **Step 1: Create a Queue**.
   - **Flink Version**: Select **1.12**.
   - **Save Job Log**: Enable this function.
   - **OBS Bucket**: Select an OBS bucket for storing job logs and grant access permissions of the OBS bucket as prompted.
   - **Enable Checkpointing**: Enable this function.
   - Enter a SQL statement in the editing pane. The following is an example. Modify the parameters in bold as you need.

     NOTE

     In this example, the syntax version of Flink OpenSource SQL is 1.12. In this example, the data source is Kafka and the result data is written to Elasticsearch.

     For details, see **PostgreSQL CDC Source Table** and **GaussDB(DWS) Result Table**.

Table 3-1 Job running parameters

Parameter	Description
Queue	A shared queue is selected by default. You can select a CCE queue with dedicated resources and configure the following parameters: **UDF Jar**: UDF Jar file. Before selecting such a file, upload the corresponding JAR file to the OBS bucket and choose **Data Management** > **Package Management** to create a package. For details, see **Creating a Package**. In SQL, you can call a UDF that is inserted into a JAR file. **NOTE** When creating a job, a sub-user can only select the queue that has been allocated to the user. If the remaining capacity of the selected queue cannot meet the job requirements, the system automatically scales up the capacity and you will be billed based on the increased capacity. When a queue is idle, the system automatically scales in its capacity.
Flink Version	The following versions are available: • **1.10**: For details about the SQL syntax, see **Flink OpenSource SQL 1.10 Syntax**. • **1.12**: For details about the SQL syntax, see **Flink OpenSource SQL 1.12 Syntax**.
CUs	Sum of the number of compute units and job manager CUs of DLI. CU is also the billing unit of DLI. One CU equals 1 vCPU and 4 GB. The value is the number of CUs required for job running and cannot exceed the number of CUs in the bound queue.
Job Manager CUs	Number of CUs of the management unit.
Parallelism	Maximum number of Flink OpenSource SQL jobs that can run at the same time. **NOTE** This value cannot be greater than four times the compute units (number of CUs minus the number of JobManager CUs).
Task Manager Configuration	Whether to set Task Manager resource parameters. If this option is selected, you need to set the following parameters: • **CU(s) per TM**: Number of resources occupied by each Task Manager. • **Slot(s) per TM**: Number of slots contained in each Task Manager.

Parameter	Description
OBS Bucket	OBS bucket to store job logs and checkpoint information. If the selected OBS bucket is not authorized, click **Authorize**.
Save Job Log	Whether to save job run logs to OBS. The logs are saved in *Bucket name*/**jobs/logs/***Directory starting with the job ID*. **CAUTION** You are advised to configure this parameter. Otherwise, no run log is generated after the job is executed. If the job fails, the run log cannot be obtained for fault locating. If this option is selected, you need to set the following parameters: **OBS Bucket**: Select an OBS bucket to store user job logs. If the selected OBS bucket is not authorized, click **Authorize**. **NOTE** If **Enable Checkpointing** and **Save Job Log** are both selected, you only need to authorize OBS once.
Alarm Generation upon Job Exception	Whether to notify users of any job exceptions, such as running exceptions or arrears, via SMS or email. If this option is selected, you need to set the following parameters: **SMN Topic** Select a user-defined SMN topic. For details about how to create a custom SMN topic, see "Creating a Topic" in **Simple Message Notification User Guide**.

Parameter	Description
Enable Checkpointing	Whether to enable job snapshots. If this function is enabled, jobs can be restored based on checkpoints. If this option is selected, you need to set the following parameters: • **Checkpoint Interval**: interval for creating checkpoints, in seconds. The value ranges from 1 to 999999, and the default value is **30**. • **Checkpoint Mode**: checkpointing mode, which can be set to either of the following values: – **At least once**: Events are processed at least once. – **Exactly once**: Events are processed only once. • **OBS Bucket**: Select an OBS bucket to store your checkpoints. If the selected OBS bucket is not authorized, click **Authorize**. Checkpoints are saved in *Bucket name***/jobs/checkpoint/***Directory starting with the job ID*. **NOTE** If **Enable Checkpointing** and **Save Job Log** are both selected, you only need to authorize OBS once.
Auto Restart upon Exception	Whether to enable automatic restart. If this function is enabled, jobs will be automatically restarted and restored when exceptions occur. If this option is selected, you need to set the following parameters: • **Max. Retry Attempts**: maximum number of retries upon an exception. The unit is times/hour. – **Unlimited**: The number of retries is unlimited. – **Limited**: The number of retries is user-defined. • **Restore Job from Checkpoint**: This parameter is available only when **Enable Checkpointing** is selected.
Idle State Retention Time	How long the state of a key is retained without being updated before it is removed in **GroupBy** or **Window**. The default value is 1 hour.
Dirty Data Policy	Policy for processing dirty data. The following policies are supported: **Ignore**, **Trigger a job exception**, and **Save**. If you set this field to **Save**, **Dirty Data Dump Address** must be set. Click the address box to select the OBS path for storing dirty data.

```
create table PostgreCdcSource(
  order_id string,
```



```
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key (order_id) not enforced
) with (
  'connector' = 'postgres-cdc',
  'hostname' = ' 192.168.15.153',--IP address of the PostgreSQL instance
  'port'= ' 5432',--Port number of the PostgreSQL instance
  'pwd_auth_name'= 'xxxxx', -- Name of the datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for the job.
  'database-name' = ' testrdsdb',--Database name of the PostgreSQL instance
  'schema-name' = ' test',-- Schema in the PostgreSQL database
  'table-name' = ' cdc_order'--Table name in the PostgreSQL database
);

create table dwsSink(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key(order_id) not enforced
) with (
  'connector' = 'gaussdb',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url'='jdbc:gaussdb://192.168.168.16:8000/testdwsdb ', ---192.168.168.16:8000 indicates the internal IP address and port of the GaussDB(DWS) instance. testdwsdb indicates the name of the created GaussDB(DWS) database.
  'table-name' = ' test\".\"dws_order', ---test indicates the schema of the created GaussDB(DWS) table, and dws_order indicates the GaussDB(DWS) table name.
  'username' = 'xxxxx',--Username of the GaussDB(DWS) instance
  'password' = 'xxxxx',--Password of the GaussDB(DWS) instance
  'write.mode' = 'insert'
);

insert into dwsSink select * from PostgreCdcSource where pay_amount > 100;
```

4. Click **Check Semantic** and ensure that the SQL statement passes the check. Click **Save**. Click **Start**, confirm the job parameters, and click **Start Now** to execute the job. Wait until the job status changes to **Running**.

Step 6: Send Data and Query Results

1. Log in to the RDS console. On the displayed page, locate the target PostgreSQL DB instance and choose **More** > **Log In** in the **Operation** column.
2. On the displayed login dialog box, enter the username and password and click **Log In**.
3. In the **Operation** column of row where the created database locates, click **SQL Window** and enter the following statement to create a table and insert data to the table:

```
insert into test.cdc_order values
('202103241000000001','webShop','2021-03-24 10:00:00','50.00','100.00','2021-03-24 10:02:03','0001','Alice','330106'),
('202103251606060001','appShop','2021-03-24 12:06:06','200.00','180.00','2021-03-24 16:10:06','0002','Jason','330106'),
('202103261000000001','webShop','2021-03-24 14:03:00','300.00','100.00','2021-03-24
```



```
10:02:03','0003','Lily','330106'),
('202103271606060001','appShop','2021-03-24 16:36:06','99.00','150.00','2021-03-24
16:10:06','0001','Henry','330106');
```

4. Connect to the created GaussDB(DWS) cluster.

   For details, see **Using the gsql CLI Client to Connect to a Cluster**.

5. Connect to the default database **testdwsdb** of a GaussDB(DWS) cluster.

   ```
   gsql -d testdwsdb -h Connection address of the GaussDB(DWS) cluster -U dbadmin -p 8000 -W password -r
   ```

6. Run the following statements to query table data:

   ```
   select * from test.dws_order;
   ```

   The query result is as follows:

   ```
   order_channel  order_channel  order_time  pay_amount  real_pay  pay_time  user_id  user_name  area_id
   202103251606060001  appShop  2021-03-24 12:06:06  200.0  180.0  2021-03-24 16:10:06  0002  Jason  330106
   202103261000000001  webShop  2021-03-24 14:03:00  300.0  100.0  2021-03-24 10:02:03  0003  Lily  330106
   ```

3.2.6 Configuring High-Reliability Flink Jobs (Automatic Restart upon Exceptions)

Scenario

If you need to configure high reliability for a Flink application, you can set the parameters when creating your Flink jobs.

Procedure

1. Create an SMN topic and add an email address or mobile number to subscribe to the topic. You will receive a subscription notification by an email or message. Click the confirmation link to complete the subscription.

   Figure 3-7 Creating a topic

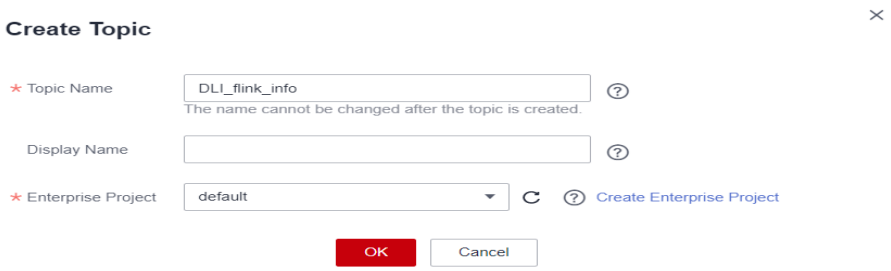
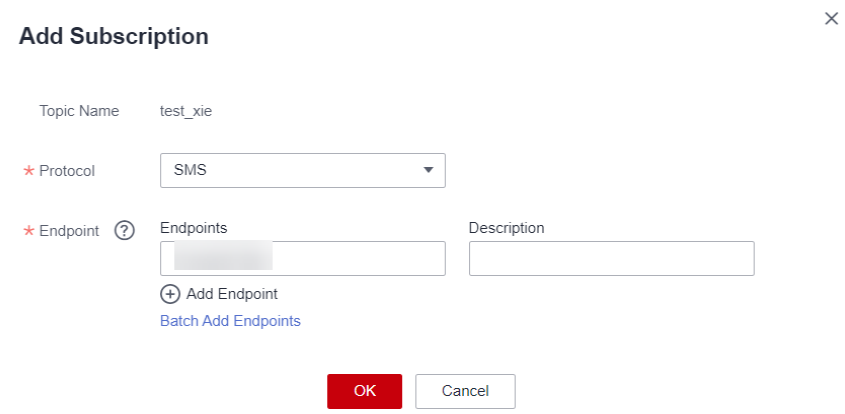


Figure 3-8 Adding a subscription



2. Log in to the DLI console, create a Flink job, write SQL statements for the job, and set running parameters. In this example, key parameters are described. Set other parameters based on your requirements.

   NOTE

   The reliability configuration of a Flink Jar job is the same as that of a SQL job, which will not be described in this section.

   a. Set **CUs**, Job **Manager CUs**, and **Max Concurrent Jobs** based on the following formulas:

      Total number of CUs = Number of manager CUs + (Total number of concurrent operators / Number of slots of a TaskManager) x Number of TaskManager CUs

      For example, with a total of 9 CUs (1 manager CU) and a maximum of 16 concurrent jobs, the number of compute-specific CUs is 8.

      If you do not configure TaskManager specifications, a TaskManager occupies 1 CU by default and has no slot. To ensure a high reliability, set the number of slots of the TaskManager to 2, according to the preceding formula.

      Set the maximum number of concurrent jobs be twice the number of CUs.

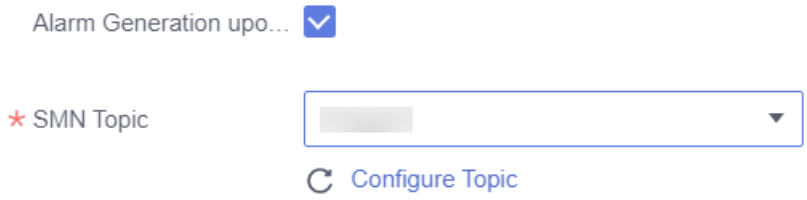
   b. Select **Save Job Log** and select an OBS bucket. If you are not authorized to access the bucket, click **Authorize**. This allows job logs be saved to your OBS bucket. If a job fails, the logs can be used for fault locating.

      Figure 3-9 Save job log

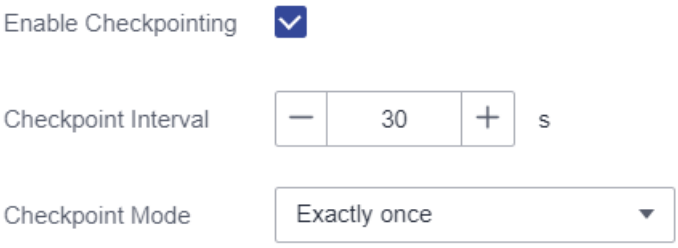
   c. Select **Alarm Generation upon Job Exception** and select the SMN topic created in **1**. This allows DLI to send notifications to your email box or phone when a job exception occurs, so you can be notified of any exceptions in time.

Figure 3-10 Alarm generation upon job exception



d. Select **Enable Checkpointing** and set the checkpoint interval and mode as needed. This function ensures that a failed Flink task can be restored from the latest checkpoint.

Figure 3-11 Checkpoint parameters



NOTE

- Checkpoint interval indicates the interval between two triggers. Checkpointing hurts real-time computing performance. To minimize the performance loss, you need to allow for the recovery duration when configuring the interval. It is recommended that the checkpoint interval **be greater than the checkpointing duration**. The recommended value is 5 minutes.
- The **Exactly once** mode ensures that each piece of data is consumed only once, and the **At least once** mode ensures that each piece of data is consumed at least once. Select a mode as you need.

e. Select **Auto Restart upon Exception** and **Restore Job from Checkpoint**, and set the number of retry attempts as needed.

f. Configure **Dirty Data Policy**. You can select **Ignore**, **Trigger a job exception**, or **Save** based on your service requirements.

g. Select a queue, and then submit and run the job.

3. Log in to the Cloud Eye console. In the navigation pane on the left, choose **Cloud Service Monitoring** > **Data Lake Insight**. Locate the target Flink job and click **Create Alarm Rule**.

   DLI provides various monitoring metrics for Flink jobs. You can define alarm rules as required using different monitoring metrics for fine-grained job monitoring.

   For details about the monitoring metrics, see **DLI Monitoring Metrics** in the *Data Lake Insight User Guide*.

3.3 Flink Jar Job Examples

Overview

You can perform secondary development based on Flink APIs to build your own Jar packages and submit them to the DLI queues to interact with data sources such as MRS Kafka, HBase, Hive, HDFS, GaussDB(DWS), and DCS.

This section describes how to interact with MRS through a custom job.

For more sample code, see the **DLI sample code**.

Environment Preparations

1. Log in to the MRS management console and create an MRS cluster. During the creation, enable **Kerberos Authentication** and select **Kafka**, **HBase**, and **HDFS**. For details about how to create an MRS cluster, see "Buying a Custom Cluster" in **MapReduce Service User Guide**.
2. Enable the UDP/TCP port in the security group rule. For details, see "Adding a Security Group Rule" in **Virtual Private Cloud User Guide**.
3. Log in to **MRS Manager**.
   a. Create a machine-machine account. Ensure that you have the **hdfs_admin** and **hbase_admin** permissions. Download the user authentication credentials, including the **user.keytab** and **krb5.conf** files.

      NOTE

      The **.keytab** file of a human-machine account becomes invalid when the user password expires. Use a machine-machine account for configuration.

   b. Click **Services**, download the client, and click **OK**.
   c. Download the configuration files from the MRS node, including **hbase-site.xml** and **hiveclient.properties**.
4. Create an elastic resource pool and queues.

   Elastic resource pools and queues provide compute resources needed to run DLI jobs. Create an elastic resource pool and queues by referring to **Creating an Elastic Resource Pool** and **Adding a Queue**.
5. Set up an enhanced datasource connection between the DLI dedicated queue and the MRS cluster and configure security group rules based on the site requirements.

   For details about how to create an enhanced datasource connection, see **Enhanced Datasource Connections** in the *Data Lake Insight User Guide*.

   For details about how to configure security group rules, see "Security Group" in **Virtual Private Cloud User Guide**.
6. Obtain the IP address and domain name mapping of all nodes in the MRS cluster, and configure the host mapping in the host information of the DLI cross-source connection.

   For details about how to add an IP-domain mapping, see **Modifying the Host Information** in the *Data Lake Insight User Guide*.

> NOTE
>
> If the Kafka server listens on the port using **hostname**, you need to add the mapping between the hostname and IP address of the Kafka Broker node to the DLI queue. Contact the Kafka service deployment personnel to obtain the hostname and IP address of the Kafka Broker node.

Prerequisites

- An elastic resource pool and queues have been created.
- When running a Flink Jar job, you need to build the secondary development application code into a JAR package and upload it to the created OBS bucket. On the DLI console, choose **Data Management** > **Package Management** to create a package. For details, see **Creating a Package**.

  > NOTE
  >
  > DLI does not support the download function. If you need to modify the uploaded data file, edit the local file and upload it again.

- Flink dependencies have been built in the DLI server and security hardening has been performed based on the open-source community version.

  To avoid issues with package compatibility or problems with logging output and dumping, be sure to exclude the following files when packaging:

  - Built-in dependencies (or set the package dependency scope to **provided** in Maven or SBT)
  - Log configuration files (for example, **log4j.properties** or **logback.xml**)
  - JAR packages for log output implementation (for example, **log4j**)

- The bucket for uploading custom configurations to OBS must be created under the master account.
- When using Flink 1.15, users need to configure agencies themselves to avoid potential impacts on job execution.

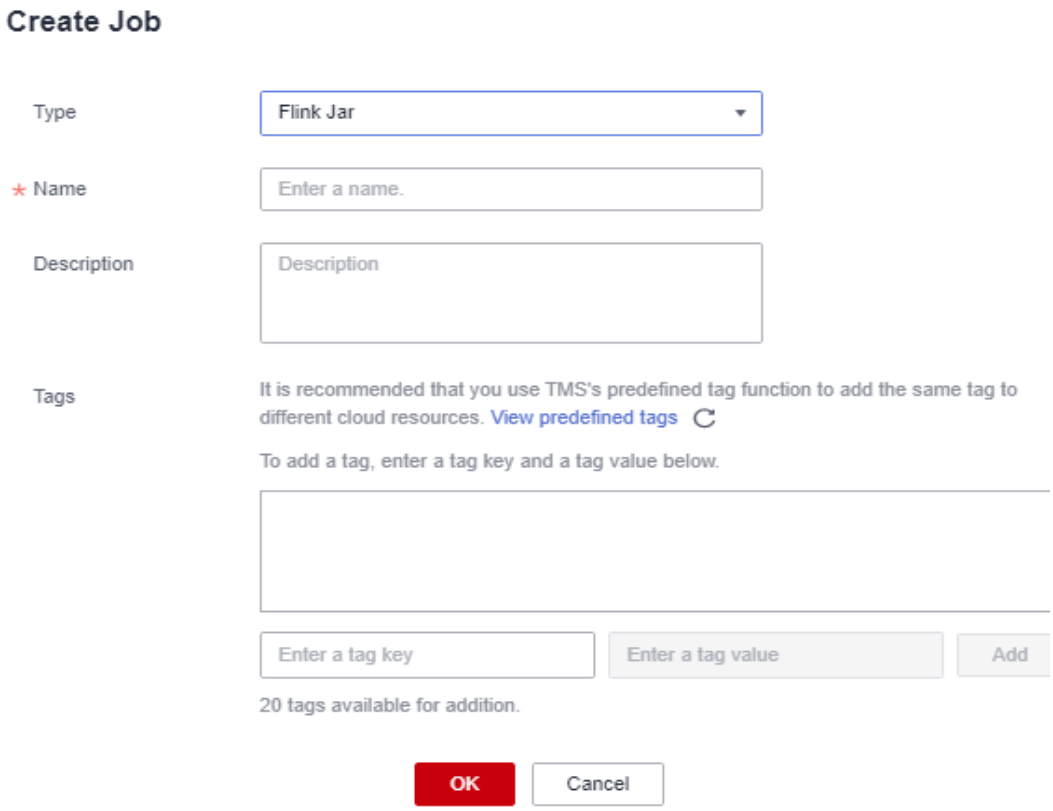
  For details, see **Customizing DLI Agency Permissions**.

How to Use

Step 1 In the left navigation pane of the DLI management console, choose **Job Management** > **Flink Jobs**. The **Flink Jobs** page is displayed.

Step 2 In the upper right corner of the **Flink Jobs** page, click **Create Job**.

Figure 3-12 Creating a Flink Jar job



Step 3 Configure job parameters.

Table 3-2 Job parameters

Parameter	Description
Type	Select **Flink Jar**.
Name	Job name, which contains 1 to 57 characters and consists of only letters, digits, hyphens (-), and underscores (_). **NOTE** The job name must be globally unique.
Description	Description of the job, which contains 0 to 512 characters.

Parameter	Description
Tags	Tags are used to identify cloud resources. A tag pair includes **Tag key** and **Tag value**. If you want to use the same tag to identify multiple cloud resources, that is, to select the same tag from the drop-down list box for all services, you are advised to create predefined tags on the Tag Management Service (TMS). For details, see **Tag Management Service User Guide**. **NOTE** • A maximum of 20 tags can be added. • Only one tag value can be added to a tag key. • **Tag key**: Enter a tag key name in the text box. **NOTE** – A tag key contains a maximum of 36 characters. The first and last characters cannot be spaces. The following characters are not allowed: =*,<>|/ – If there are predefined tags, you can select a tag from the drop-down list box. • **Tag value**: Enter a tag value in the text box. **NOTE** – A tag value contains a maximum of 43 characters. The first and last characters cannot be spaces. The following characters are not allowed: =*,<>|/ – If there are predefined tags, you can select a tag from the drop-down list box.

Step 4 Click **OK** to enter the **Edit** page.

Step 5 Select a queue. A Flink Jar job can run only on general queues.

Figure 3-13 Selecting a queue

Step 6 Upload the JAR package.

The Flink version must be the same as that specified in the JAR package.

Figure 3-14 Uploading the JAR package

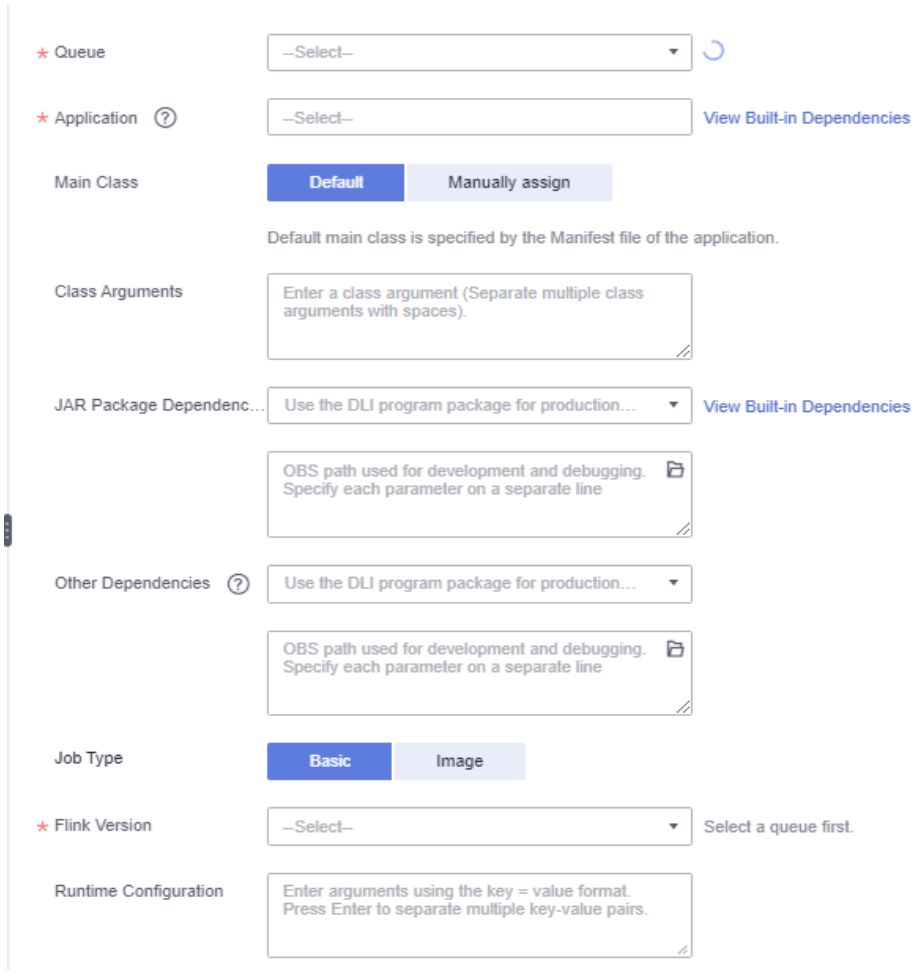


Table 3-3 Description

Parameter	Description
Application	User-defined package. Before selecting a package, upload the corresponding JAR package to the OBS bucket and create a package on the **Data Management** > **Package Management** page. For details, see **Creating a Package**.
Main Class	Name of the main class of the JAR package to be loaded, for example, **KafkaMessageStreaming**. • **Default**: The value is specified based on the **Manifest** file in the JAR package. • **Manually assign**: You must enter the class name and confirm the class arguments (separate arguments with spaces). **NOTE** When a class belongs to a package, the package path must be carried, for example, **packagePath.KafkaMessageStreaming**.
Class Arguments	List of arguments of a specified class. The arguments are separated by spaces.

Parameter	Description
JAR Package Dependencies	User-defined dependencies. Before selecting a package, upload the corresponding JAR package to the OBS bucket and create a JAR package on the **Data Management** > **Package Management** page. For details, see **Creating a Package**.
Other Dependencies	User-defined dependency files. Before selecting a file, upload the corresponding file to the OBS bucket and create a package of any type on the **Data Management** > **Package Management** page. For details, see **Creating a Package**. You can add the following content to the application to access the corresponding dependency file: **fileName** indicates the name of the file to be accessed, and **ClassName** indicates the name of the class that needs to access the file. `ClassName.class.getClassLoader().getResource("userData/fileName")`
Job Type	This parameter is displayed when the queue type is CCE. • **Basic** • **Image**: Select the image name and image version. Images are set on the Software Repository for Container (SWR) console.
Flink Version	Before selecting a Flink version, you need to select the queue to which the Flink version belongs. Flink 1.15 is recommended.

Step 7 Configure job parameters.

CAUTION

Minimal submission means Flink only submits the necessary job dependencies, not the entire Flink environment. By setting the scope of non-Connector Flink dependencies (starting with **flink-**) and third-party libraries (like Hadoop, Hive, Hudi, and MySQL-CDC) to **provided**, you ensure these dependencies are excluded from the Jar job, avoiding conflicts with Flink core dependencies.

- Only Flink 1.15 supports minimal submission of Flink Jar jobs. Enable this by configuring **flink.dli.job.jar.minimize-submission.enabled=true** in the runtime optimization parameters.
- For Flink-related dependencies, use the **provided** scope by adding **<scope>provided</scope>** in the dependencies, especially for non-Connector dependencies under the **org.apache.flink** group starting with **flink-**.
- For dependencies related to Hadoop, Hive, Hudi, and MySQL-CDC, also use the **provided** scope by adding **<scope>provided</scope>** in the dependencies.
- In the Flink source code, only methods marked with **@Public** or **@PublicEvolving** are intended for user invocation. DLI guarantees compatibility with these methods.

Figure 3-15 Configuring parameters

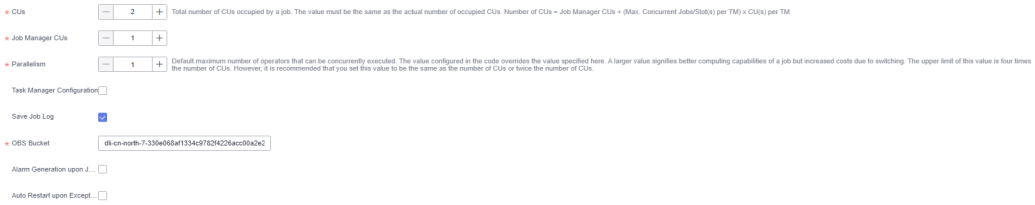


Table 3-4 Parameter description

Parameter	Description
CUs	One CU has one vCPU and 4 GB memory. The number of CUs ranges from 2 to 400.
Job Manager CUs	Set the number of CUs on a management unit. The value ranges from 1 to 4. The default value is **1**.
Parallelism	Maximum number of parallel operators in a job. **NOTE** • The value must be less than or equal to four times the number of compute units (CUs minus the number of job manager CUs). • You are advised to set this parameter to a value greater than that configured in the code. Otherwise, job submission may fail.
Task Manager Configuration	Whether to set Task Manager resource parameters. If this option is selected, you need to set the following parameters: • **CU(s) per TM**: Number of resources occupied by each Task Manager. • **Slot(s) per TM**: Number of slots contained in each Task Manager.
Save Job Log	Whether to save the job running logs to OBS. If this option is selected, you need to set the following parameters: **OBS Bucket**: Select an OBS bucket to store user job logs. If the selected OBS bucket is not authorized, click **Authorize**.
Alarm Generation upon Job Exception	Whether to report job exceptions, for example, abnormal job running or exceptions due to an insufficient balance, to users via SMS or email. If this option is selected, you need to set the following parameters: **SMN Topic** Select a user-defined SMN topic. For details about how to create a custom SMN topic, see "Creating a Topic" in **Simple Message Notification User Guide**.

Parameter	Description
Auto Restart upon Exception	Whether to enable automatic restart. If this function is enabled, jobs will be automatically restarted and restored when exceptions occur. If this option is selected, you need to set the following parameters: • **Max. Retry Attempts**: maximum number of retry times upon an exception. The unit is **Times/hour**. – **Unlimited**: The number of retries is unlimited. – **Limited**: The number of retries is user-defined. • **Restore Job from Checkpoint**: Restore the job from the latest checkpoint. If you select this parameter, you also need to set **Checkpoint Path**. **Checkpoint Path**: Select the checkpoint saving path. The value must be the same as the checkpoint path you set in the application package. Note that the checkpoint path for each job must be unique. Otherwise, the checkpoint cannot be obtained.

Step 8 Click **Save** on the upper right of the page.

Step 9 Click **Start** on the upper right side of the page. On the displayed **Start Flink Job** page, confirm the job specifications and the price, and click **Start Now** to start the job.

After the job is started, the system automatically switches to the **Flink Jobs** page, and the created job is displayed in the job list. You can view the job status in the **Status** column. After a job is successfully submitted, the job status will change from **Submitting** to **Running**. After the execution is complete, the message **Completed** is displayed.

If the job status is **Submission failed** or **Running exception**, the job submission failed or the job did not execute successfully. In this case, you can move the cursor over the status icon in the **Status** column of the job list to view the error details. You can click to copy these details. After handling the fault based on the provided information, resubmit the job.

NOTE

Other buttons are as follows:

Save As: Save the created job as a new job.

----End

Related Operations

- **How Do I Configure Job Parameters?**
  a. In the Flink job list, select your desired job.
  b. Click **Edit** in the **Operation** column.
  c. Configure the parameters as needed.

List of parameters of a specified class. The parameters are separated by spaces.

Parameter input format: --Key 1 Value 1 --Key 2 Value 2

For example, if you enter the following parameters on the console:

--bootstrap.server 192.168.168.xxx:9092

The parameters are parsed by ParameterTool as follows:

Figure 3-16 Parsed parameters

- **How Do I View Job Logs?**
  a. In the Flink job list, click the name of a desired job to access its details page.
  b. Click the **Run Log** tab and view job logs on the console.

     Only the latest run logs are displayed. For more information, see the OBS bucket that stores logs.

3.4 Writing Data to OBS Using Flink Jar

Overview

DLI allows you to use a custom JAR package to run Flink jobs and write data to OBS. This section describes how to write processed Kafka data to OBS. You need to modify the parameters in the example Java code based on site requirements.

Environment Preparations

Development tools such as IntelliJ IDEA and other development tools, JDK, and Maven have been installed and configured.

NOTE

- For details about how to configure the pom.xml file of the Maven project, see "POM file configurations" in **Java Example Code (Flink 1.12)**.
- Ensure that you can access the public network in the local compilation environment.

Constraints

- In the left navigation pane of the DLI console, choose **Global Configuration** > **Service Authorization**. On the page displayed, select **Tenant Administrator(Global service)** and click **Update**.
- The bucket to which data is written must be an OBS bucket created by a main account.
- When using Flink 1.15, users need to configure agencies themselves to avoid potential impacts on job execution.

  For details, see **Customizing DLI Agency Permissions**.

Java Example Code (Flink 1.15)

- **POM** file configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.huaweicloud</groupId>
        <artifactId>dli-flink-demo</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <groupId>org.example</groupId>
    <artifactId>flink-1.15-demo</artifactId>
    <properties>
        <flink.version>1.15.0</flink.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-statebackend-rocksdb</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-table-planner_2.12</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.14.2</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>3.3.0</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>com.huawei.dli.GetUserConfigFileDemo</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
```



```
            </plugin>
        </plugins>
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*.*</include>
                </includes>
            </resource>
        </resources>
    </build>
</project>
```

- Example code

```
package com.huawei.dli;

import com.huawei.dli.source.CustomParallelSource;

import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.EmbeddedRocksDBStateBackend;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.OnCheckpointRollingPolicy;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;

public class GetUserConfigFileDemo {
    private static final Logger LOG = LoggerFactory.getLogger(GetUserConfigFileDemo.class);

    public static void main(String[] args) {
        try {
            ParameterTool params = ParameterTool.fromArgs(args);
            LOG.info("Params: " + params.toString());

            StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();

            // set checkpoint
            String checkpointPath = params.get("checkpoint.path", "obs://bucket/checkpoint/
jobId_jobName/");
            LocalDateTime localDateTime = LocalDateTime.ofEpochSecond(System.currentTimeMillis() /
1000,
                0, ZoneOffset.ofHours(8));
            String dt = localDateTime.format(DateTimeFormatter.ofPattern("yyyyMMdd_HH:mm:ss"));
            checkpointPath = checkpointPath + dt;

            streamEnv.setStateBackend(new EmbeddedRocksDBStateBackend());
            streamEnv.getCheckpointConfig().setCheckpointStorage(checkpointPath);
            streamEnv.getCheckpointConfig().setExternalizedCheckpointCleanup(
                CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
            streamEnv.enableCheckpointing(30 * 1000);

            DataStream<String> stream = streamEnv.addSource(new CustomParallelSource())
                .setParallelism(1)
                .disableChaining();

            String outputPath = params.get("output.path", "obs://bucket/outputPath/jobId_jobName");
```



```
        // Get user dependents config
        URL url = GetUserConfigFileDemo.class.getClassLoader().getResource("userData/user.config");
        if (url != null) {
            Path filePath = org.apache.flink.util.FileUtils.absolutizePath(new Path(url.getPath()));
            try {
                String config = org.apache.flink.util.FileUtils.readFileUtf8(new File(filePath.getPath()));
                LOG.info("config is {}", config);
                // Do something by config
            } catch (IOException e) {
                LOG.error(e.getMessage(), e);
            }
        }

        // Sink OBS
        final StreamingFileSink<String> sinkForRow = StreamingFileSink
            .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
            .withRollingPolicy(OnCheckpointRollingPolicy.build())
            .build();

        stream.addSink(sinkForRow);

        streamEnv.execute("sinkForRow");
    } catch (Throwable e) {
        LOG.error(e.getMessage(), e);
    }
  }
}
```

Table 3-5 Parameter description

Parameter	Description	Example
output.path	OBS path to which data will be written	obs://bucket/output
checkpoint.path	Checkpoint OBS path	obs://bucket/checkpoint

Java Example Code (Flink 1.12)

- **POM** file configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <parent>
        <artifactId>Flink-demo</artifactId>
        <groupId>com.huaweicloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>flink-kafka-to-obs</artifactId>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!--Flink version-->
        <flink.version>1.12.2</flink.version>
        <!--JDK version -->
        <java.version>1.8</java.version>
```

```
    <!--Scala 2.11 -->
    <scala.binary.version>2.11</scala.binary.version>
    <slf4j.version>2.13.3</slf4j.version>
    <log4j.version>2.10.0</log4j.version>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
</properties>

<dependencies>
    <!-- flink -->
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-java</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- kafka -->
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka_2.11</artifactId>
        <version>${flink.version}</version>
    </dependency>

    <!-- logging -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>${slf4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-jcl</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.3.0</version>
```



```
            <executions>
               <execution>
                  <phase>package</phase>
                  <goals>
                     <goal>single</goal>
                  </goals>
               </execution>
            </executions>
            <configuration>
               <archive>
                  <manifest>
                     <mainClass>com.huaweicloud.dli.FlinkKafkaToObsExample</mainClass>
                  </manifest>
               </archive>
               <descriptorRefs>
                  <descriptorRef>jar-with-dependencies</descriptorRef>
               </descriptorRefs>
            </configuration>
         </plugin>
      </plugins>
      <resources>
         <resource>
            <directory>../main/config</directory>
            <filtering>true</filtering>
            <includes>
               <include>**/*.*</include>
            </includes>
         </resource>
      </resources>
   </build>
</project>
```

- Example code

```
import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend;
import org.apache.flink.core.fs.Path;
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import
org.apache.flink.streaming.api.functions.sink.filesystem.bucketassigners.DateTimeBucketAssigner;
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.OnCheckpointRollingPolicy;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;

/**
 * @author xxx
 * @date 6/26/21
 */
public class FlinkKafkaToObsExample {
    private static final Logger LOG = LoggerFactory.getLogger(FlinkKafkaToObsExample.class);

    public static void main(String[] args) throws Exception {
        LOG.info("Start Kafka2OBS Flink Streaming Source Java Demo.");
        ParameterTool params = ParameterTool.fromArgs(args);
        LOG.info("Params: " + params.toString());

        // Kafka connection address
        String bootstrapServers;
        // Kafka consumer group
        String kafkaGroup;
```



```
        // Kafka topic
        String kafkaTopic;
        // Consumption policy. This policy is used only when the partition does not have a checkpoint or the checkpoint expires.
        // If a valid checkpoint exists, consumption continues from this checkpoint.
        // When the policy is set to LATEST, the consumption starts from the latest data. This policy will ignore the existing data in the stream.
        //        When the policy is set to EARLIEST, the consumption begins with the earliest available data in the stream. This policy ensures that all valid data in the stream is obtained.
        String offsetPolicy;
        // OBS file output path, in the format of obs://bucket/path.
        String outputPath;
        // Checkpoint output path, in the format of obs://bucket/path.
        String checkpointPath;

        bootstrapServers = params.get("bootstrap.servers", "xxxx:9092,xxxx:9092,xxxx:9092");
        kafkaGroup = params.get("group.id", "test-group");
        kafkaTopic = params.get("topic", "test-topic");
        offsetPolicy = params.get("offset.policy", "earliest");
        outputPath = params.get("output.path", "obs://bucket/output");
       checkpointPath = params.get("checkpoint.path", "obs://bucket/checkpoint");

        try {
            //Create an execution environment.
            StreamExecutionEnvironment streamEnv = StreamExecutionEnvironment.getExecutionEnvironment();
            streamEnv.setParallelism(4);
            RocksDBStateBackend rocksDbBackend = new RocksDBStateBackend(checkpointPath, true);
            RocksDBStateBackend rocksDbBackend = new RocksDBStateBackend(new FsStateBackend(checkpointPath), true);
            streamEnv.setStateBackend(rocksDbBackend);
            // Enable Flink checkpointing mechanism. If enabled, the offset information will be synchronized to Kafka.
            streamEnv.enableCheckpointing(300000);
            // Set the minimum interval between two checkpoints.
            streamEnv.getCheckpointConfig().setMinPauseBetweenCheckpoints(60000);
            // Set the checkpoint timeout duration.
            streamEnv.getCheckpointConfig().setCheckpointTimeout(60000);
            // Set the maximum number of concurrent checkpoints.
            streamEnv.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
            // Retain checkpoints when a job is canceled.
            streamEnv.getCheckpointConfig().enableExternalizedCheckpoints(
                    CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

            // Source: Connect to the Kafka data source.
            Properties properties = new Properties();
            properties.setProperty("bootstrap.servers", bootstrapServers);
            properties.setProperty("group.id", kafkaGroup);
            properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, offsetPolicy);
            String topic = kafkaTopic;

            // Create a Kafka consumer.
            FlinkKafkaConsumer<String> kafkaConsumer =
                    new FlinkKafkaConsumer<>(topic, new SimpleStringSchema(), properties);
            /**
             * Read partitions from the offset submitted by the consumer group (specified by group.id in the consumer attribute) in Kafka brokers.
             * If the partition offset cannot be found, set it by using the auto.offset.reset parameter.
             * For details, see https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/datastream/kafka/.
             */
            kafkaConsumer.setStartFromGroupOffsets();

            // Add Kafka to the data source.
            DataStream<String> stream = streamEnv.addSource(kafkaConsumer).setParallelism(3).disableChaining();

            // Create a file output stream.
            final StreamingFileSink<String> sink = StreamingFileSink
```



```
                // Specify the file output path and row encoding format.
                .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
                // Specify the file output path and bulk encoding format. Files are output in parquet
format.
                //.forBulkFormat(new Path(outputPath), ParquetAvroWriters.forGenericRecord(schema))
                // Specify a custom bucket assigner.
                .withBucketAssigner(new DateTimeBucketAssigner<>())
                // Specify the rolling policy.
                .withRollingPolicy(OnCheckpointRollingPolicy.build())
                .build();

        // Add sink for DIS Consumer data source
        stream.addSink(sink).disableChaining().name("obs");

        // stream.print();
        streamEnv.execute();
    } catch (Exception e) {
        LOG.error(e.getMessage(), e);
    }
  }
}
```

Table 3-6 Parameter description

Parameter	Description	Example
bootstrap.servers	Kafka connection address	*IP address of the Kafka service* 1:9092, *IP address of the Kafka service* 2:9092, *IP address of the Kafka service* 3:9092
group.id	Kafka consumer group	**test-group**
topic	Kafka consumption topic	**test-topic**
offset.policy	Kafka offset policy	**earliest**
output.path	OBS path to which data will be written	obs://bucket/output
checkpoint.path	Checkpoint OBS path	obs://bucket/checkpoint

Compiling and Running the Application

After the application is developed, upload the JAR package to DLI by referring to **Flink Jar Job Examples** and check whether related data exists in the OBS path.

3.5 Using Flink Jar to Connect to Kafka that Uses SASL_SSL Authentication

Overview

Use Flink Jar to connect to a Kafka with SASL_SSL authentication enabled.

For details about how to use Flink OpenSource SQL to connect to Kafka with SASL_SSL authentication enabled, see **Kafka Source Table**.

Environment Preparations

- You have purchased a general-purpose queue on the DLI console.
- You have purchased a Kafka instance and enabled SASL_SSL authentication.
- You have created an enhanced datasource connection on DLI and have bound it to the queue that can communicate with Kafka.

Notes

- To connect to Kafka over SASL_SSL, you need to specify the path of the **truststore** file in the properties for both consumers and producers.
- The consumer and producer are initialized in **taskmanager**. You need to obtain the path of the **truststore** file of the container corresponding to the **taskmanager** and import the file to the properties file for the initialization.
- Kafka source can be introduced by the **open** method.

  Figure 3-17 Obtaining Kafka source

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.DeserializationSchema;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

import java.util.Properties;

public class SourceKafkaConsumer<T> extends FlinkKafkaConsumer<T> {

    public SourceKafkaConsumer(String topic, DeserializationSchema<T> valueDeserializer, Properties props) {
        super(topic, valueDeserializer, props);
    }

    @Override
    public void open(Configuration configuration) throws Exception {
        String jksPath = SourceKafkaConsumer.class.getClassLoader().getResource( name: "userData/client.jks").getPath();
        super.properties.setProperty("ssl.truststore.location", jksPath);
        super.open(configuration);
    }
}
```

- Kafka sink can be introduced by the **initializeState** method.

Figure 3-18 Obtaining Kafka sink

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.SerializationSchema;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;

import java.util.Properties;

public class SinkKafkaProducer<IN> extends FlinkKafkaProducer<IN>{

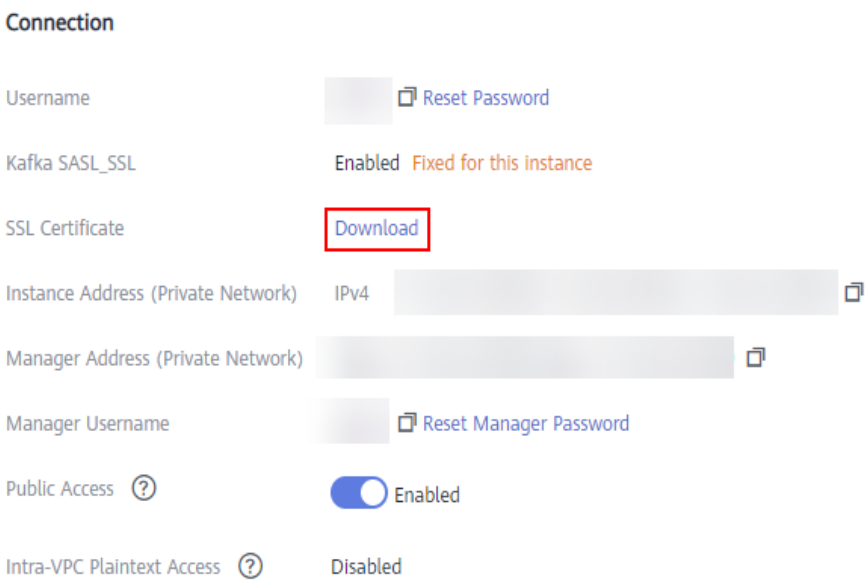
    public SinkKafkaProducer(String topicId, SerializationSchema<IN> serializationSchema, Properties producerConfig) {
        super(topicId, serializationSchema, producerConfig);
    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {
        String jksPath = SinkKafkaProducer.class.getClassLoader().getResource( name: "userData/client.jks").getPath();
        producerConfig.setProperty("ssl.truststore.location", jksPath);
        super.initializeState(context);
    }
}
```

Procedure

Step 1 Download the SSL certificate from the basic information page of the Kafka instance, decompress the certificate, and upload the **clinet.jks** file to OBS.

Figure 3-19 Downloading SSL certificate

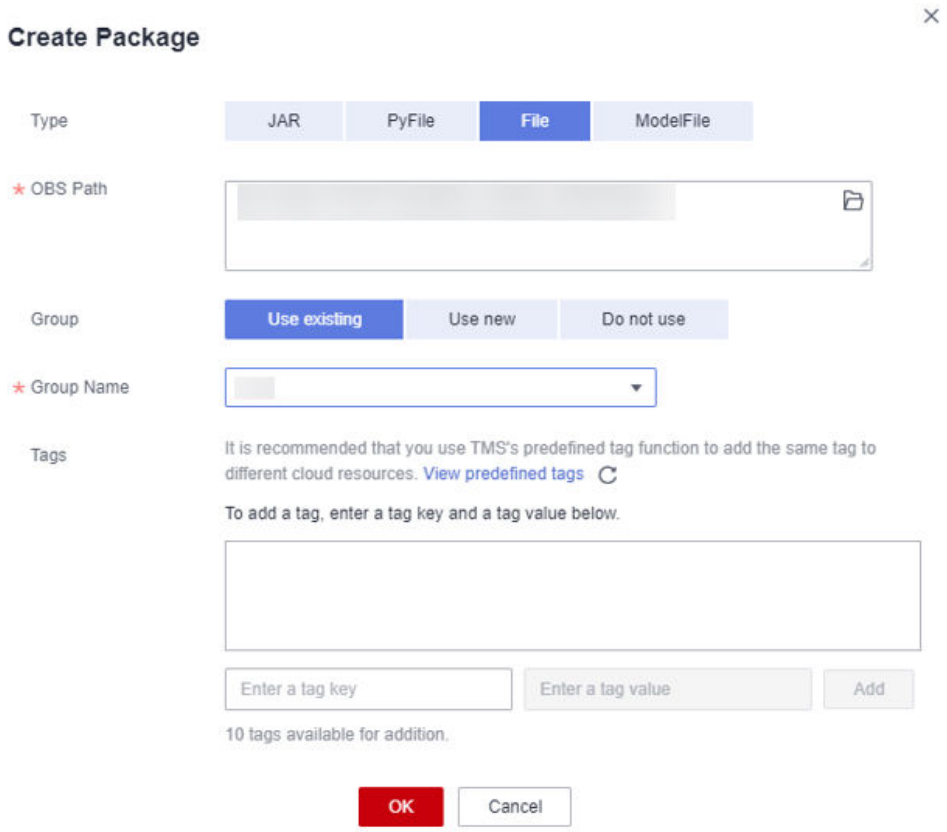


Step 2 On the DLI console, choose **Data Management** > **Package Management** in the left navigation pane. On the displayed page, click **Create** to create the **clinet.jks** package.

The required parameters are as follows:

- **Type**: Select **File**.
- **OBS Path**: Specify the OBS path of the **clinet.jks** file.
- **Group Name**: Enter a name for a new group or select an existing group name.

Figure 3-20 Creating **clinet.jks** package on DLI

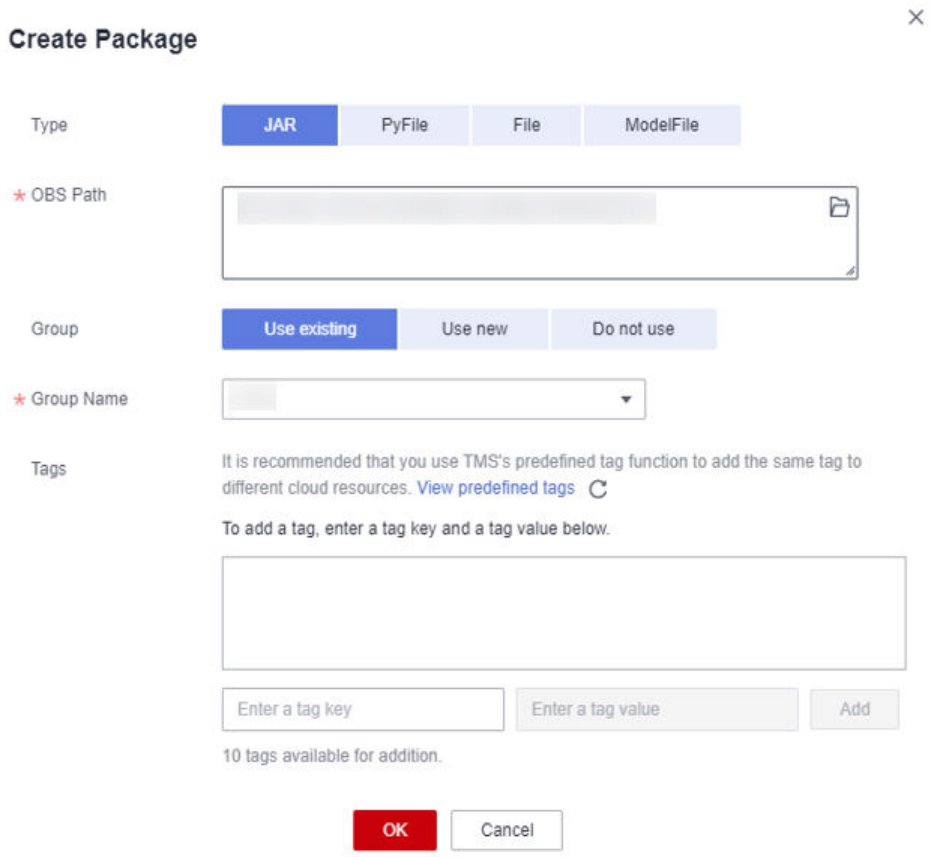


Step 3 Package the sample code. On the DLI console, choose **Data Management** > **Package Management**. Click **Create** on the displayed page to create a Flink JAR package. For details about the sample code, see **Kafaka ToKafakaExample.java**, **SinkKafkaProducer.java**, and **SourceKafkaConsumer.java**.

The parameters are as follows:

- **Type**: Select **JAR**.
- **OBS Path**: Specify the OBS path of the **Flink Jar**.
- **Group Name**: Enter a name for a new group or select an existing group name.

Figure 3-21 Creating a Flink JAR package

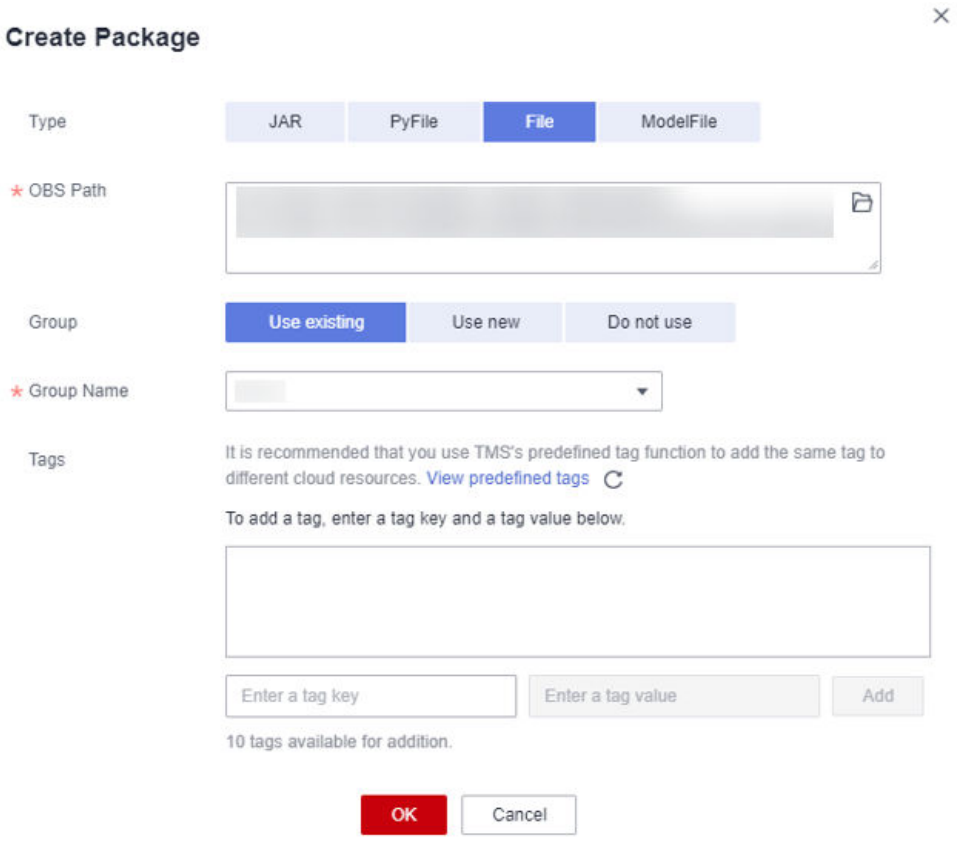


Step 4 On the DLI console, choose **Data Management** > **Package Management** in the navigation pane on the left. On the displayed page, click **Create** to create the **KafkaToKafka.properties** package. For the sample file, see **KafkaToKafkaExample.properties**.

The parameters are as follows:

- **Type**: Select **File**.
- **OBS Path**: Specify the OBS path of the **KafkaToKafka.properties** file.
- **Group Name**: Enter a name for a new group or select an existing group name.

Figure 3-22 Creating a DLI package



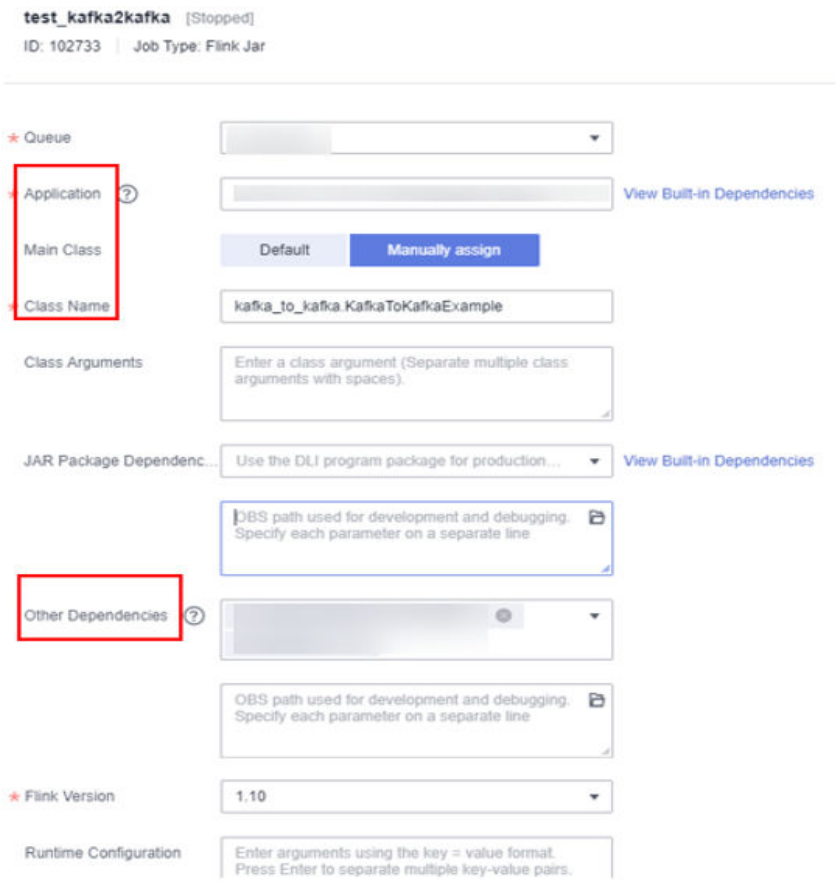
Step 5 Create a Flink Jar job and run it.

Import the JAR imported in **Step 3** and other dependencies to the Flink Jar job, and specify the main class.

The parameters are as follows:

- **Queue**: Select the queue where the job will run.
- **Application**: Select a custom program.
- **Main Class**: Select **Manually assign**.
- **Class Name**: Enter the class name and class arguments (separate arguments with spaces).
- **Other Dependencies**: Select custom dependencies. Select the **.jks** and properties files imported in the **Step 2** and **Step 4**.
- **Flink Version**: Select **1.10**.

Figure 3-23 Creating a Flink Jar job



Step 6 Verify the result.

When the job is in the running state, send data to **kafka source.topic** to check whether **kafka sink.topic** can receive the data. If the data can be received, the connection is successful.

Figure 3-24 Viewing job tasks

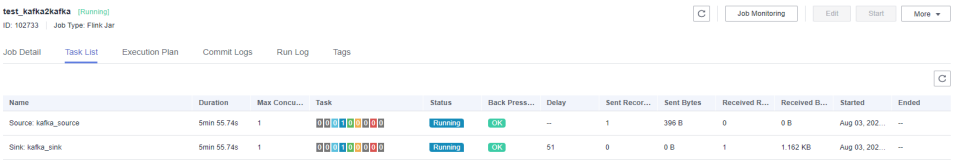
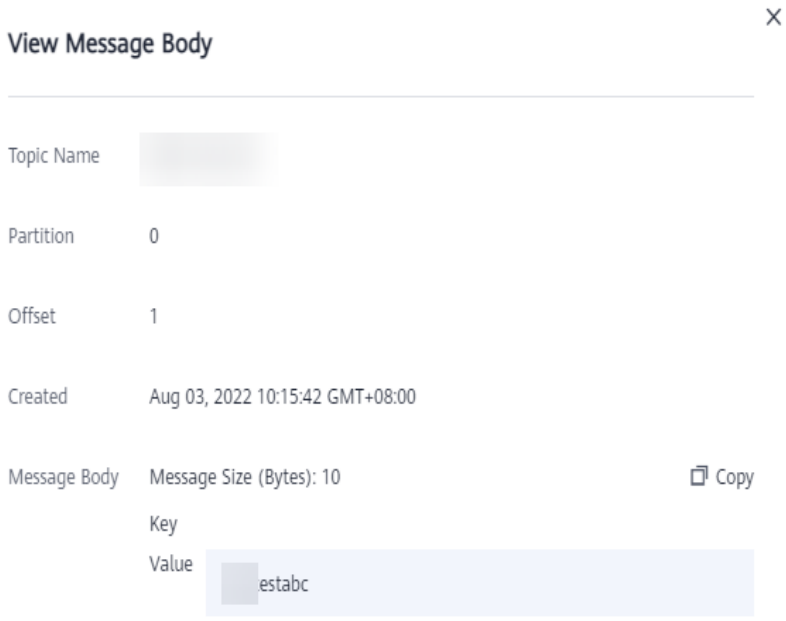


Figure 3-25 Viewing kafka sink.topic



----End

Example Java Code

- **POM** file configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <parent>
        <artifactId>Flink-demo</artifactId>
        <groupId>com.huaweicloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>flink-kafka-to-obs</artifactId>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!--Flink version-->
        <flink.version>1.12.2</flink.version>
        <!--JDK version -->
        <java.version>1.8</java.version>
        <!--Scala 2.11 -->
        <scala.binary.version>2.11</scala.binary.version>
        <slf4j.version>2.13.3</slf4j.version>
        <log4j.version>2.10.0</log4j.version>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- flink -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
```



```
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- kafka -->
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka_2.11</artifactId>
        <version>${flink.version}</version>
    </dependency>

    <!-- logging -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>${slf4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-jcl</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.huaweicloud.dli.FlinkKafkaToObsExample</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
```



```
                </descriptorRefs>
            </configuration>
        </plugin>
    </plugins>
    <resources>
        <resource>
            <directory>../main/config</directory>
            <filtering>true</filtering>
            <includes>
                <include>**/*.*</include>
            </includes>
        </resource>
    </resources>
  </build>
</project>
```

- KafkaToKafkaExample.java

  userData is a fixed file path name, which cannot be changed or customized.

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;


public class KafkaToKafkaExample {
    private static final Logger LOG = LoggerFactory.getLogger(KafkaToKafkaExample.class);

    public static void main(String[] args) throws Exception {
        LOG.info("Start Kafka2Kafka Flink Streaming Source Java Demo.");
        String propertiesPath = KafkaToKafkaExample.class.getClassLoader()
            .getResource("userData/KafkaToKafka.properties").getPath();
        ParameterTool params = ParameterTool.fromPropertiesFile(propertiesPath);
        LOG.info("Params: " + params.toString());

        // Kafka connection address
        String bootstrapServers;
        // Kafka consumer group
        String kafkaGroup;
        // Kafka topic
        String sourceTopic;
        String sinkTopic;
        // Consumption policy. This policy is used only when the partition does not have a checkpoint or the checkpoint expires.
        // If a valid checkpoint exists, consumption continues from this checkpoint.
        // When the policy is set to LATEST, the consumption starts from the latest data. This policy will ignore the existing data in the stream.
        //         When the policy is set to EARLIEST, the consumption begins with the earliest available data in the stream. This policy ensures that all valid data in the stream is obtained.
        String offsetPolicy;
        //SASL_SSL configuration items. Set the JAAS username and password, which are the username and password entered when SASL_SSL is enabled when the Kafka instance is created,
        // or those set when the SASL_SSL user is created. The format is as follows:
        // org.apache.kafka.common.security.plain.PlainLoginModule required
        // username="yourUsername"
        // password="yourPassword";
        String saslJaasConfig;
        // Checkpoint output path, which is in the format of obs://bucket/path.
        String checkpointPath;

        bootstrapServers = params.get("bootstrap.servers", "xxx:9093,xxx:9093,xxx:9093");
```



```
        kafkaGroup = params.get("source.group", "test-group");
        sourceTopic = params.get("source.topic", "test-source-topic");
        sinkTopic = params.get("sink.topic", "test-sink-topic");
        offsetPolicy = params.get("offset.policy", "earliest");
        saslJaasConfig = params.get("sasl.jaas.config",
            "org.apache.kafka.common.security.plain.PlainLoginModule"
                + "required\nusername=\"yourUsername\"\npassword=\"yourPassword\";");
        checkpointPath = params.get("checkpoint.path", "obs://bucket/path");


        try {
            // Create the execution environment.
            StreamExecutionEnvonment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
            RocksDBStateBackend rocksDbBackend = new RocksDBStateBackend(checkpointPath, true);
            streamEnv.setStateBackend(rocksDbBackend);
            // Enable Flink checkpointing mechanism. If enabled, the offset information will be
synchronized to Kafka.
            streamEnv.enableCheckpointing(300000);
            // Set the minimum interval between two checkpoints.
            streamEnv.getCheckpointConfig().setMinPauseBetweenCheckpoints(60000);
            // Set the checkpoint timeout duration.
            streamEnv.getCheckpointConfig().setCheckpointTimeout(60000);
            // Set the maximum number of concurrent checkpoints.
            streamEnv.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
            // Retain checkpoints when a job is canceled.
            streamEnv.getCheckpointConfig().enableExternalizedCheckpoints(
                CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

            // Connect to the Kafka data source.
            Properties sourceProperties = new Properties();
            sourceProperties.setProperty("bootstrap.servers", bootstrapServers);
            sourceProperties.setProperty("group.id", kafkaGroup);
            sourceProperties.setProperty("sasl.jaas.config", saslJaasConfig);
            sourceProperties.setProperty("sasl.mechanism", "PLAIN");
            sourceProperties.setProperty("security.protocol", "SASL_SSL");
            sourceProperties.setProperty("ssl.truststore.password", "dms@kafka");
            sourceProperties.setProperty("ssl.endpoint.identification.algorithm", "");
            sourceProperties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, offsetPolicy);


            // Create a Kafka consumer.
            SourceKafkaConsumer<String> kafkaConsumer =
                new SourceKafkaConsumer<>(sourceTopic, new SimpleStringSchema(), sourceProperties);
            /**
             * Read partitions from the offset submitted by the consumer group (specified by group.id in
the consumer attribute) in Kafka brokers.
             * If the partition offset cannot be found, set it by using the auto.offset.reset parameter.
             * For details, see https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/connectors/
datastream/kafka/.
             */
            kafkaConsumer.setStartFromGroupOffsets();

            // Add Kafka to the data source.
            DataStream<String> stream =
streamEnv.addSource(kafkaConsumer).name("kafka_source").setParallelism(1).disableChaining();

            // Connect to the Kafka data sink.
            Properties sinkProperties = new Properties();
            sinkProperties.setProperty("bootstrap.servers", bootstrapServers);
            sinkProperties.setProperty("sasl.jaas.config", saslJaasConfig);
            sinkProperties.setProperty("sasl.mechanism", "PLAIN");
            sinkProperties.setProperty("security.protocol", "SASL_SSL");
            sinkProperties.setProperty("ssl.truststore.password", "dms@kafka");
            sinkProperties.setProperty("ssl.endpoint.identification.algorithm", "");

            // Create a Kafka producer.
            SinkKafkaProducer<String> kafkaProducer = new SinkKafkaProducer<>(sinkTopic, new
SimpleStringSchema(),
```



```
                sinkProperties);

            // Add Kafka to the data sink.
            stream.addSink(kafkaProducer).name("kafka_sink").setParallelism(1).disableChaining();

            // stream.print();
            streamEnv.execute();
        } catch (Exception e) {
            LOG.error(e.getMessage(), e);
        }
    }
}
```

- SinkKafkaProducer.java

  userData is a fixed file path name, which cannot be changed or customized.

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.SerializationSchema;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;

import java.util.Properties;

public class SinkKafkaProducer<IN> extends FlinkKafkaProducer<IN>{

    public SinkKafkaProducer(String topicId, SerializationSchema<IN> serializationSchema, Properties
producerConfig) {
        super(topicId, serializationSchema, producerConfig);
    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {
        String jksPath = SinkKafkaProducer.class.getClassLoader().getResource("userData/
client.jks").getPath();
        producerConfig.setProperty("ssl.truststore.location", jksPath);
        super.initializeState(context);
    }
}
```

- SourceKafkaConsumer.java

  userData is a fixed file path name, which cannot be changed or customized.

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.DeserializationSchema;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

import java.util.Properties;

public class SourceKafkaConsumer<T> extends FlinkKafkaConsumer<T> {

    public SourceKafkaConsumer(String topic, DeserializationSchema<T> valueDeserializer, Properties
props) {
        super(topic, valueDeserializer, props);
    }

    @Override
    public void open(Configuration configuration) throws Exception {
        String jksPath = SourceKafkaConsumer.class.getClassLoader().getResource("userData/
client.jks").getPath();
        super.properties.setProperty("ssl.truststore.location", jksPath);
        super.open(configuration);
    }
}
```

- KafkaToKafkaExample.properties

```
bootstrap.servers=xxx:9093,xxx:9093,xxx:9093
checkpoint.path=obs://bucket/path
source.group=swq-test-group
```



```
source.topic=topic-swq
sink.topic=topic-swq-out
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required\n\
 username="xxxx"\n\
 password="xxxx";
```

3.6 Using Flink Jar to Read and Write Data from and to DIS

Overview

Read and write data from and to DIS using a Flink Jar job based on Flink 1.12.

In Flink 1.12, Flink Opensource SQL jobs cannot use **connector** to read and write data from and to DIS. So, you are advised to use the method described in this section.

DIS is no longer recommended for Flink 1.15. You are advised to use DMS Kafka instead. For details, see **Kafka connector**.

Environment Preparations

- You have purchased a general-purpose queue on the DLI console.
- You have purchased a DIS stream by referring to **Creating a DIS Stream**.
- If Flink 1.12 is used, the DIS connector version must be 2.0.1 or later. For details about the code, see **DIS Flink Connector Dependencies**. For details about how to configure the connector, see **Customizing a Flink Streaming Job**.

  NOTE

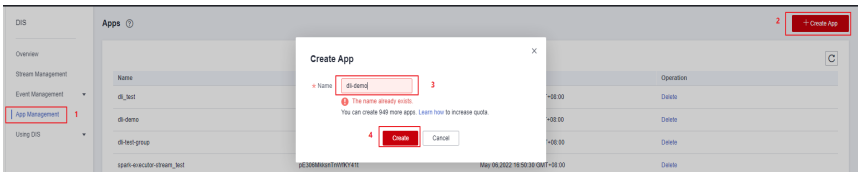
  If data is read from DIS and **groupId** is configured, you need to create the required app name on the **App Management** page of the DIS console in advance.
- Do not place the **disToDis.properties** file in the generated JAR file. The code contains the path of the file. If the file is placed in the JAR file, the code cannot find the path of the file.

Procedure

1. Create a DIS stream. For details, see **Creating a DIS Stream**.

   In the left navigation pane of the DIS console, choose **App Management**. On the page displayed, click **Create App**. In the **Create App** dialog box, set **Name** to the value of **groupId**.

   Figure 3-26 Create App

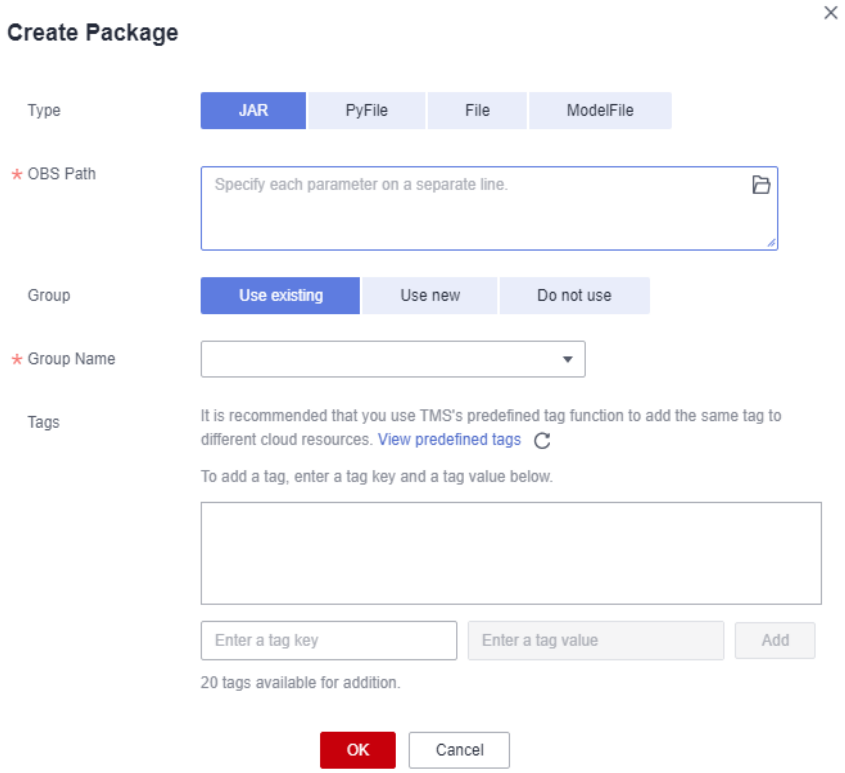
2. Create a Flink JAR file.

In the left navigation pane of the DLI console, choose **Data Management** > **Package Management**. On the page displayed, click **Create Package** to create a Flink JAR file. For details about the example code, see **FlinkDisToDisExample.java**.

Table 3-7 Main parameters for creating a Flink JAR file

Parameter	Description	Example Value
Type	Package type. Possible values are as follows: • **JAR**: JAR file • **PyFile**: User Python file • **File**: User file • **ModelFile**: User AI model file	JAR
OBS Path	Select the OBS path of the corresponding package. **NOTE** • The program package must be uploaded to OBS in advance. • Only files can be selected.	OBS path where Flink Jar is stored
Group Name	• If **Use existing** is selected for **Group**, select an existing group. • If **Use new** is selected for **Group**, enter a custom group name. • If **Do not use** is selected for **Group**, this parameter is unavailable.	Enter a custom group name or select an existing group name.

Figure 3-27 Creating a Flink JAR file



3. Create the distoDis package.

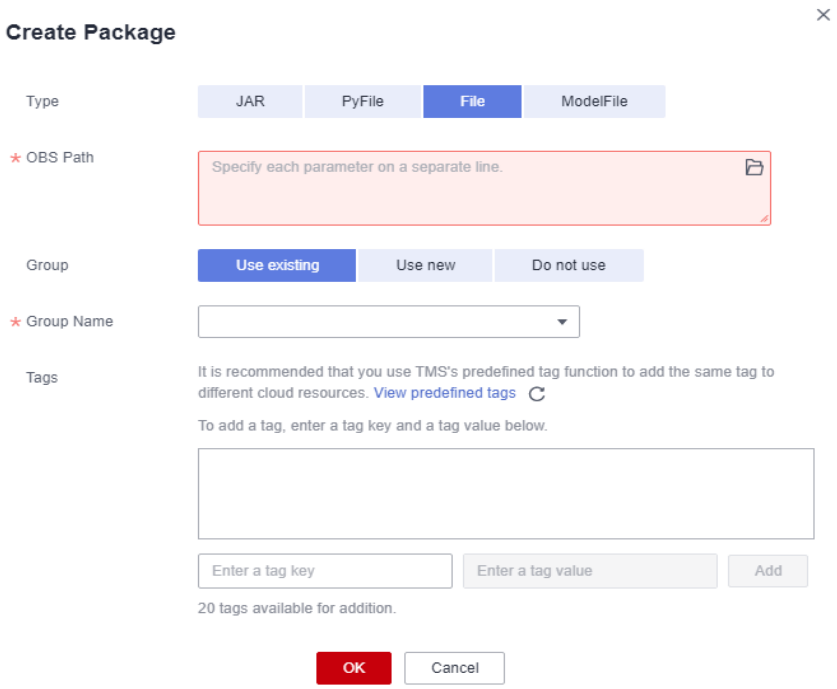
   In the left navigation pane of the DLI console, choose **Data Management** > **Package Management**. On the page displayed, click **Create Package** to create the package corresponding to the **disToDis.properties** file. For details about the example code, see **Example disToDis.properties**.

   Table 3-8 Main parameters for creating the package corresponding to disToDis.properties

Parameter	Description	Example Value
Type	Package type. Possible values are as follows: • **JAR**: JAR file • **PyFile**: User Python file • **File**: User file • **ModelFile**: User AI model file	File
OBS Path	Select the OBS path of the corresponding package. **NOTE** • The program package must be uploaded to OBS in advance. • Only files can be selected.	OBS path where **disToDis.properties** is stored.

Parameter	Description	Example Value
Group Name	• If **Use existing** is selected for **Group**, select an existing group. • If **Use new** is selected for **Group**, enter a custom group name. • If **Do not use** is selected for **Group**, this parameter is unavailable.	Enter a custom group name or select an existing group name.

Figure 3-28 Creating the package corresponding to disToDis.properties



4. Create a Flink Jar job and run it.

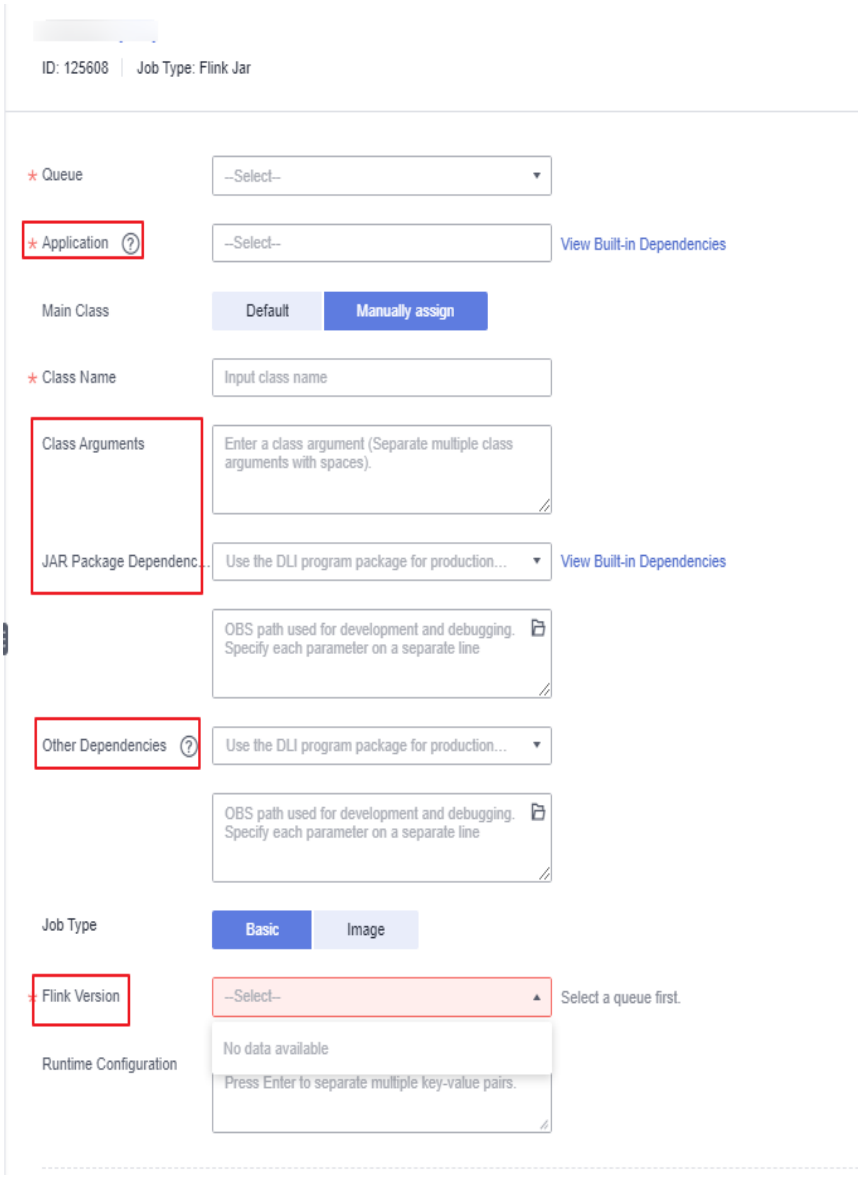
   For details about how to create a Flink Jar job, see **Creating a Flink Jar Job**. Select the Flink Jar file created in **2** in the application, select the **properties** file created in **3** in other dependency files, and specify the main class.

Table 3-9 Parameters for creating a Flink Jar job

Parameter	Description	Example Value
Queue	**NOTE** • A Flink Jar job can run only on a pre-created dedicated queue. • If no dedicated queue is available in the **Queue** drop-down list, create a dedicated queue and bind it to the current user.	Select the queue where the job will run.

Parameter	Description	Example Value
Application	User-defined package.	Custom program package
Main Class	Name of the JAR file to be loaded, for example, **FlinkDisToDisExample**. Possible values are as follows: • **Default**: The value is specified based on the **Manifest** file in the JAR file. • **Manually assign**: You must enter the class name and confirm the class arguments (separated by spaces). **NOTE** When a class belongs to a package, the main class path must contain the complete package path, for example, **packagePath.KafkaMessageStreaming**.	Manually assign
Other Dependencies	User-defined dependency files. Other dependency files need to be referenced in the code. Before selecting a dependency file, upload the file to the OBS bucket and choose **Data Management** > **Package Management** to create a package. The package type is not limited. For details, see **Creating a Package**. You can add the following command to the application to access the corresponding dependency file. **fileName** indicates the name of the file to be accessed, and **ClassName** indicates the name of the class that needs to access the file. `ClassName.class.getClassLoader().getResource("userData/fileName")`	Select the **properties** file in **3**.
Flink Version	Before selecting a Flink version, you need to select the queue to which the Flink version belongs. Currently, 1.10, 1.11, and 1.12 are supported.	1.12

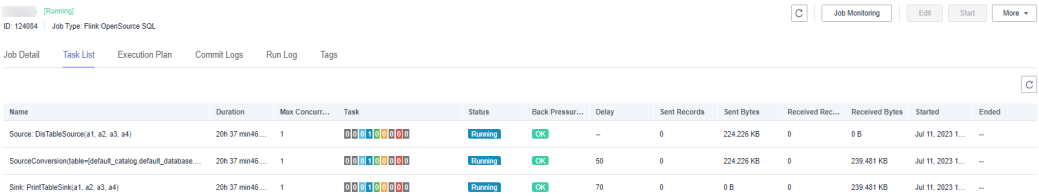
Figure 3-29 Creating a Flink Jar job



5. Verify the result.

When a job is in the **Running** state, send data to the DIS source stream to check whether the DIS sink stream can receive the data. If there are bytes sent and received, it indicates that data is received.

Figure 3-30 Viewing the verification result



Example Java Code

- Dependencies of DIS Flink connector

```
<dependency>
        <groupId>com.huaweicloud.dis</groupId>
        <artifactId>huaweicloud-dis-flink-connector_2.11</artifactId>
        <version>2.0.1</version>
        <exclusions>
            <exclusion>
                <groupId>org.apache.flink</groupId>
                <artifactId>*</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org.apache.logging.log4j</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
```

- **POM** file configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Flink-dis-12</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!--Flink version-->
        <flink.version>1.12.2</flink.version>
        <!--JDK version -->
        <java.version>1.8</java.version>
        <!--Scala 2.11 -->
        <scala.binary.version>2.11</scala.binary.version>
        <slf4j.version>2.13.3</slf4j.version>
        <log4j.version>2.10.0</log4j.version>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- dis -->
        <dependency>
            <groupId>com.huaweicloud.dis</groupId>
            <artifactId>huaweicloud-dis-flink-connector_2.11</artifactId>
            <version>2.0.1</version>
            <exclusions>
                <exclusion>
                    <groupId>org.apache.flink</groupId>
                    <artifactId>*</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.apache.logging.log4j</groupId>
                    <artifactId>*</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <!-- flink -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
```



```
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- logging -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>${slf4j.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-jcl</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.flink.FlinkDisToDisExample</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </plugin>
    </plugins>
```



```
        <resources>
            <resource>
                <directory>../main/config</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*.*</include>
                </includes>
            </resource>
        </resources>
    </build>
</project>
```

- FlinkDisToDisExample.java example(see **Customizing a Flink Streaming Job**)

```
package com.flink;
import com.huaweicloud.dis.DISConfig;
import com.huaweicloud.dis.adapter.common.consumer.DisConsumerConfig;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend;
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.dis.FlinkDisConsumer;
import org.apache.flink.streaming.connectors.dis.FlinkDisProducer;
import org.apache.flink.util.TernaryBoolean;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Collections;

/**
 * Read data from dis and then write them into another dis channel.
 */
public class FlinkDisToDisExample {
    private static final Logger LOG = LoggerFactory.getLogger(FlinkDisToDisExample.class);

    public static void main(String[] args) throws IOException {
        LOG.info("Read data from dis and write them into dis.");
        String propertiesPath = FlinkDisToDisExample.class.getClassLoader()
            .getResource("userData/disToDis.properties").getPath();
        ParameterTool params = ParameterTool.fromPropertiesFile(propertiesPath);

        // DIS endpoint, for example, https://dis.cn-north-1.myhuaweicloud.com
        String endpoint = params.get("disEndpoint");
        // ID of the region where DIS is, for example, cn-north-1
        String region = params.get("region");
        // AK of the user
        String ak = params.get("ak");
        // SK of the user
        String sk = params.get("sk");
        // Project ID of the user
        String projectId = params.get("projectId");
        // Name of the DIS source stream
        String sourceChannel = params.get("sourceChannel");
        // Name of the DIS sink stream
        String sinkChannel = params.get("sinkChannel");
        // Consumption policy. This policy is used only when the partition has no checkpoint or the checkpoint has expired. If a valid checkpoint exists, the consumption continues from this checkpoint.
        // When the policy is set to LATEST, the consumption starts from the latest data. This policy will ignore the existing data in the stream.
        //         When the policy is set to EARLIEST, the consumption begins with the earliest available data in the stream. This policy ensures that all valid data in the stream is obtained.
        String startingOffsets = params.get("startOffset");
        // Consumer group ID. Different clients in the same consumer group can consume the same stream at the same time.
        String groupId = params.get("groupId");
```



```
        // Checkpoint output path, which is in the format of obs://bucket/path/.
        String checkpointBucket = params.get("checkpointPath");

        // DIS Config
        DISConfig disConfig = DISConfig.buildDefaultConfig();
        disConfig.put(DISConfig.PROPERTY_ENDPOINT, endpoint);
        disConfig.put(DISConfig.PROPERTY_REGION_ID, region);
        disConfig.put(DISConfig.PROPERTY_AK, ak);
        disConfig.put(DISConfig.PROPERTY_SK, sk);
        disConfig.put(DISConfig.PROPERTY_PROJECT_ID, projectId);
        disConfig.put(DisConsumerConfig.AUTO_OFFSET_RESET_CONFIG, startingOffsets);
        disConfig.put(DisConsumerConfig.GROUP_ID_CONFIG, groupId);
        // Whether to proactively update segment information and the update interval (ms). If proactive
scaling is required, enable this function.
        disConfig.put(FlinkDisConsumer.KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS, "10000");

        SimpleDateFormat dateTimeFormat = new SimpleDateFormat("yyyy-MM-dd_HH-mm-ss");
        String time = dateTimeFormat.format(System.currentTimeMillis());
        String checkpointPath = checkpointBucket + time;

        try {
            StreamExecutionEnvironment env =
                StreamExecutionEnvironment.getExecutionEnvironment();
            // Set the checkpoint.
            RocksDBStateBackend rocksDBStateBackend = new RocksDBStateBackend(new
FsStateBackend(checkpointPath),
                TernaryBoolean.TRUE);
            env.setStateBackend(rocksDBStateBackend);
            // Enable Flink checkpointing. If enabled, the offset information will be synchronized to Kafka.
            env.enableCheckpointing(180000);
            // Set the minimum interval between two checkpoints.
            env.getCheckpointConfig().setMinPauseBetweenCheckpoints(60000);
            // Set the checkpoint timeout duration.
            env.getCheckpointConfig().setCheckpointTimeout(60000);
            // Set the maximum number of concurrent checkpoints.
            env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
            // Retain checkpoints when a job is canceled.
            env.getCheckpointConfig().enableExternalizedCheckpoints(
                CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

            FlinkDisConsumer<String> consumer =
                new FlinkDisConsumer<>(
                    Collections.singletonList(sourceChannel), new SimpleStringSchema(), disConfig);
            DataStream<String> sourceStream = env.addSource(consumer, "disSource");

            FlinkDisProducer<String> producer = new FlinkDisProducer<>(
                sinkChannel, new SimpleStringSchema(), disConfig);
            sourceStream.addSink(producer).disableChaining().name("dis-to-dis");

            env.execute();
        } catch (Exception ex) {
            LOG.error(ex.getMessage(), ex);
        }
    }
}
```

- Example disToDis.properties (Note: The value of **groupId** is the name of the app created in step 1.)

  NOTE

  Hard-coded or plaintext AK and SK pose significant security risks. To ensure security, encrypt your AK and SK, store them in configuration files or environment variables, and decrypt them when needed.

```
# Endpoint of the site where DIS is deployed, for example, https://dis.cn-
north-1.myhuaweicloud.com
disEndpoint=xx
# ID of the region where DIS is, for example, cn-north-1
region=xx
```



```
# AK of the user
ak=xx
# SK of the user
sk=xx
# Example ID: 6m3nhAGTLxmNfZ4HOit
projectId=xx
# DIS source stream, for example, OpenSource_outputmTtkR
sourceChannel=xx
# DIS sink stream, for example, OpenSource_disQFXD
sinkChannel=xx
# DIS consumer group, which needs to be created in DIS App Management page in advance.
groupId=xx
# Consumption mode. The value can be EARLIEST or LATEST.
startOffset=LATEST
# Flink path for storing checkpoints
checkpointPath=obs://bucket/path/
```

FAQ

- Q: Why does a job fail to be executed and the run log contain the following error information?

  ```
  java.lang.NoSuchMethodError: org.apache.flink.api.java.ClosureCleaner.clean(Ljava/lang/Object;Z)V
  ```

  A: Because the huaweicloud-dis-flink-connector_2.11 version you selected is too early. Select 2.0.1 or later.

- Q: Why cannot my job read data from DIS and does the TaskManager run log contain the following error information?

  ```
  ERROR com.huaweicloud.dis.adapter.common.consumer.Coordinator      [] - Failed to
  getCheckpointAsync, error : [400 : {"errorCode":"DIS.4332","message":"app not found. "}], request :
  [{"stream_name":"xx","partition_id":"shardId-0000000000","checkpoint_type":"LAST_READ","app_nam
  e":"xx"}]
  ```

  A: Because **group.id** used to read data from DIS is not created on the DIS **Application Management** page in advance.

3.7 Flink Job Agencies

3.7.1 Flink OpenSource SQL Jobs Using DEW to Manage Access Credentials

Scenario

When DLI writes the output data of Flink jobs to MySQL or GaussDB(DWS), you need to set attributes such as the username and password in the connector. However, information such as usernames and passwords is highly sensitive and needs to be encrypted to ensure user data privacy.

Data Encryption Workshop (DEW) and Cloud Secret Management Service (CSMS) joint form a secure, reliable, and easy-to-use privacy data encryption and decryption solution.

Users or applications can use CSMS to create, retrieve, update, and delete credentials in a unified manner throughout the secret lifecycle. CSMS can help you eliminate risks incurred by hardcoding, plaintext configuration, and permission abuse.

This section walks you through on how to use DEW to manage access credentials for Flink OpenSource SQL jobs.

Prerequisites

- A shared secret has been created on the DEW console and the secret value has been stored. For details, see **Creating a Shared Secret**.
- An agency has been created and authorized for DLI to access DEW. The agency must have been granted the following permissions:
  - Permission of the **ShowSecretVersion** interface for querying secret versions and secret values in DEW: **csms:secretVersion:get**.
  - Permission of the **ListSecretVersions** interface for listing secret versions in DEW: **csms:secretVersion:list**.
  - Permission to decrypt DEW secrets: **kms:dek:decrypt**

  For details about agency permission examples, see **Customizing DLI Agency Permissions** and **Agency Permission Policies in Common Scenarios**.
- DEW can be used to manage access credentials only in Flink 1.15. When creating a Flink job, select version 1.15 and configure the information of the agency that allows DLI to access DEW for the job.
- On the DLI management console, create an enhanced datasource connection and configure the network connection between DLI and the data source.

  For details, see **Enhanced Datasource Connections**.

Syntax

```
create table tableName(
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression)
)
with (
   ...
  'dew.endpoint'='',
  'dew.csms.secretName'='',
  'dew.csms.decrypt.fields'='',
  'dew.projectId'='',
  'dew.csms.version'=''

);
```

Parameter Description

Table 3-10 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
dew.endpoint	Yes	None	String	Endpoint of the DEW service to be used. See **Regions and Endpoints**. Configuration example: **'dew.endpoint'='kms.cn-xxxx.myhuaweicloud.com'**

Parameter	Mandatory	Default Value	Data Type	Description
dew.projectId	No	Yes	String	ID of the project DEW belongs to. The default value is the ID of the project where the Flink job is. See **Obtaining a Project ID**.
dew.csms.secretName	Yes	None	String	Name of the shared secret in DEW's secret management. Configuration example: **'dew.csms.secretName'='***secretInfo***'**
dew.csms.decrypt.fields	Yes	None	String	Specify which fields in the **connector with** attribute need to be decrypted using DEW's CSMS. Separate the field attributes with commas, for example, **'dew.csms.decrypt.fields'='field1,field2,field3'**
dew.csms.version	No	Latest version	String	Version number (certificate version identifier) of the shared secret in DEW's secret management. If not specified, the latest version of the shared secret is obtained by default. Configuration example: **'dew.csms.version'='v1'**

Example

This example demonstrates how to configure Flink OpenSource SQL to manage access credentials using DEW by generating random data through a DataGen table and outputting it to a MySQL result table.

1. Create an enhanced datasource connection between DLI and MySQL.
2. Create an agency for DLI to access DEW and complete authorization. For details, see **Customizing DLI Agency Permissions**.
3. Create a shared secret in DEW. For details, see **Creating a Shared Secret**.
   a. Log in to the DEW management console.
   b. In the navigation pane on the left, choose **Cloud Secret Management Service** > **Secrets**.
   c. On the displayed page, click **Create Secret**. Set basic secret information.
   d. In this example, the MySQL credential value is configured as follows:
      - "MySQLUsername":"demo"
      - "MySQLPassword":"*******", where ******* indicates the password for accessing the MySQL database.

4. Enter a SQL statement in the editing pane. The following is an example:

```
create table dataGenSource(
  user_id string,
  amount int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --Generate a piece of data per second.
  'fields.user_id.kind' = 'random', --Specify a random generator for the user_id field.
  'fields.user_id.length' = '3' --Limit the length of user_id to 3.
);

CREATE TABLE jdbcSink (
  user_id string,
  amount int
 )
WITH (
 'connector' = 'jdbc',
  'url? = 'jdbc:mysql://MySQLAddress:MySQLPort/flink',--flink is the MySQL database where the orders table locates.
 'table-name' = 'orders',
 'username' = 'MySQLUsername',  -- Shared secret in DEW whose name is secretInfo and version is v1. The key MySQLUsername defines the secret value. The value is the user's sensitive information.
 'password' = 'MySQLPassword',  -- Shared secret in DEW whose name is secretInfo and version is v1. The key MySQLPassword defines the secret value. The value is the user's sensitive information.
 'sink.buffer-flush.max-rows' = '1',
 'dew.endpoint'='kms.cn-xxxx.myhuaweicloud.com', --Endpoint information for the DEW service being used
 'dew.csms.secretName'='secretInfo', --Name of the DEW shared secret
 'dew.csms.decrypt.fields'='username,password', --The password field value must be decrypted and replaced using DEW secret management.
 'dew.csms.version'='v1'
);

insert into jdbcSink select * from dataGenSOurce;
```

3.7.2 Flink Jar Jobs Using DEW to Acquire Access Credentials for Reading and Writing Data from and to OBS

Scenario

To write the output data of a Flink Jar job to OBS, AK/SK is required for accessing OBS. To ensure the security of AK/SK data, you can use Data Encryption Workshop (DEW) and Cloud Secret Management Service (CSMS) for unified management of AK/SK, effectively avoiding sensitive information leakage and business risks caused by hard-coded or plaintext configuration of programs.

This section walks you through on how a Flink Jar job acquires an AK/SK to read and write data from and to OBS.

Prerequisites

- A shared secret has been created on the DEW console and the secret value has been stored. For details, see **Creating a Shared Secret**.
- An agency has been created and authorized for DLI to access DEW. The agency must have been granted the following permissions:
  - Permission of the **ShowSecretVersion** interface for querying secret versions and secret values in DEW: **csms:secretVersion:get**.
  - Permission of the **ListSecretVersions** interface for listing secret versions in DEW: **csms:secretVersion:list**.

  - Permission to decrypt DEW secrets: **kms:dek:decrypt**

  For details about agency permission examples, see **Customizing DLI Agency Permissions** and **Agency Permission Policies in Common Scenarios**.

- DEW can be used to manage access credentials only in Flink 1.15. When creating a Flink job, select version 1.15 and configure the information of the agency that allows DLI to access DEW for the job. For details about how to create a custom agency and configure it, see **Customizing DLI Agency Permissions**.
- To use this function, you need to configure AK/SK for all OBS buckets.

Syntax

On the Flink Jar job editing page, set **Runtime Configuration** as needed. The configuration information is as follows:

Different OBS buckets use different AK/SK authentication information. You can use the following configuration method to specify the AK/SK information based on the bucket. For details about the parameters, see **Table 3-11**.

```
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key=USER_AK_CSMS_KEY
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key=USER_SK_CSMS_KEY
flink.hadoop.fs.obs.security.provider=com.dli.provider.UserObsBasicCredentialProvider
flink.hadoop.fs.dew.csms.secretName=CredentialName
flink.hadoop.fs.dew.endpoint=ENDPOINT
flink.hadoop.fs.dew.csms.version=VERSION_ID
flink.hadoop.fs.dew.csms.cache.time.second=CACHE_TIME
flink.dli.job.agency.name=USER_AGENCY_NAME
```

Parameter Description

Table 3-11 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key	Yes	None	String	*USER_BUCKET_NAME* needs to be replaced with the user's OBS bucket name. The value of this parameter is the key defined by the user in the CSMS shared secret. The value corresponding to the key is the user's access key ID (AK). The user must have the permission to access the bucket on OBS.

Parameter	Mandatory	Default Value	Data Type	Description
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key	Yes	None	String	*USER_BUCKET_NAME* needs to be replaced with the user's OBS bucket name. The value of this parameter is the key defined by the user in the CSMS shared secret. The value corresponding to the key is the user's secret access key (SK). The user must have the permission to access the bucket on OBS.
flink.hadoop.fs.obs.security.provider	Yes	None	String	OBS AK/SK authentication mechanism, which uses DEW-CSMS' secret management to obtain the AK and SK for accessing OBS. The default value is **com.dli.provider.UserObsBasicCredentialProvider**.
flink.hadoop.fs.dew.endpoint	Yes	None	String	Endpoint of the DEW service to be used. See **Regions and Endpoints**. Configuration example: **flink.hadoop.fs.dew.endpoint=kms.cn-xxxx.myhuaweicloud.com**
flink.hadoop.fs.dew.projectId	No	Yes	String	ID of the project DEW belongs to. The default value is the ID of the project where the Flink job is. See **Obtaining a Project ID**.
flink.hadoop.fs.dew.csms.secretName	Yes	None	String	Name of the shared secret in DEW's secret management. Configuration example: **flink.hadoop.fs.dew.csms.secretName=*secretInfo***
flink.hadoop.fs.dew.csms.version	No	Latest version	String	Version number (certificate version identifier) of the shared secret in DEW's secret management. If not specified, the latest version of the shared secret is obtained by default. Configuration example: **flink.hadoop.fs.dew.csms.version=v1**

Parameter	Mandatory	Default Value	Data Type	Description
flink.hadoop.fs.dew.csms.cache.time.second	No	3600	Long	Cache duration after the CSMS shared secret is obtained during Flink job access. The unit is second. The default value is 3600 seconds.
flink.dli.job.agency.name	Yes	-	String	Custom agency name.

Sample Code

This section describes how to write processed DataGen data to OBS. You need to modify the parameters in the sample Java code based on site requirements.

1. Create an agency for DLI to access DEW and complete authorization. For details, see **Customizing DLI Agency Permissions**.
2. Create a shared secret in DEW. For details, see **Creating a Shared Secret**.
   a. Log in to the DEW management console.
   b. In the navigation pane on the left, choose **Cloud Secret Management Service** > **Secrets**.
   c. On the displayed page, click **Create Secret**. Set basic secret information.
3. Set job parameters on the DLI Flink Jar job editing page.
   - Class Name
     ```
     com.dli.demo.dew.DataGen2FileSystemSink
     ```
   - Class Arguments
     ```
     --checkpoint.path obs://test/flink/jobs/checkpoint/120891/
     --output.path obs://dli/flink.db/79914/DataGen2FileSystemSink
     ```
   - Runtime Configuration
     ```
     flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key=USER_AK_CSMS_KEY
     flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key=USER_SK_CSMS_KEY
     flink.hadoop.fs.obs.security.provider=com.dli.provider.UserObsBasicCredentialProvider
     flink.hadoop.fs.dew.csms.secretName=obsAksK
     flink.hadoop.fs.dew.endpoint=kmsendpoint
     flink.hadoop.fs.dew.csms.version=v6
     flink.hadoop.fs.dew.csms.cache.time.second=3600
     flink.dli.job.agency.name=***
     ```
4. Flink Jar job example.
   - **Environment preparation**

     Development tools such as IntelliJ IDEA and other development tools, JDK, and Maven have been installed and configured.

     Dependency package in POM file configurations

     ```
     <properties>
         <flink.version>1.15.0</flink.version>
     </properties>
     ```



```
<dependencies>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-statebackend-rocksdb</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- fastjson -->
    <dependency>
        <artifactId>fastjson</artifactId>
        <version>2.0.15</version>
    </dependency>
</dependencies>
```

- **Sample code**

```
package com.huawei.dli.demo.dew;

import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.EmbeddedRocksDBStateBackend;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.OnCheckpointRollingPolicy;
import org.apache.flink.streaming.api.functions.source.ParallelSourceFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.Random;

public class DataGen2FileSystemSink {
    private static final Logger LOG = LoggerFactory.getLogger(DataGen2FileSystemSink.class);

    public static void main(String[] args) {
        ParameterTool params = ParameterTool.fromArgs(args);
        LOG.info("Params: " + params.toString());
        try {
            StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();

            // set checkpoint
            String checkpointPath = params.get("checkpoint.path", "obs://bucket/checkpoint/
jobId_jobName/");
            LocalDateTime localDateTime =
LocalDateTime.ofEpochSecond(System.currentTimeMillis() / 1000,
                0, ZoneOffset.ofHours(8));
            String dt =
localDateTime.format(DateTimeFormatter.ofPattern("yyyyMMdd_HH:mm:ss"));
            checkpointPath = checkpointPath + dt;

            streamEnv.setStateBackend(new EmbeddedRocksDBStateBackend());
            streamEnv.getCheckpointConfig().setCheckpointStorage(checkpointPath);
            streamEnv.getCheckpointConfig().setExternalizedCheckpointCleanup(
                CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
            streamEnv.enableCheckpointing(30 * 1000);
```



```
            DataStream<String> stream = streamEnv.addSource(new DataGen())
                .setParallelism(1)
                .disableChaining();

            String outputPath = params.get("output.path", "obs://bucket/outputPath/
jobId_jobName");

            // Sink OBS
            final StreamingFileSink<String> sinkForRow = StreamingFileSink
                .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
                .withRollingPolicy(OnCheckpointRollingPolicy.build())
                .build();

            stream.addSink(sinkForRow);

            streamEnv.execute("sinkForRow");
        } catch (Exception e) {
            LOG.error(e.getMessage(), e);
        }
    }
}

class DataGen implements ParallelSourceFunction<String> {

    private boolean isRunning = true;

    private Random random = new Random();

    @Override
    public void run(SourceContext<String> ctx) throws Exception {
        while (isRunning) {
            JSONObject jsonObject = new JSONObject();
            jsonObject.put("id", random.nextLong());
            jsonObject.put("name", "Molly" + random.nextInt());
            jsonObject.put("address", "hangzhou" + random.nextInt());
            jsonObject.put("birthday", System.currentTimeMillis());
            jsonObject.put("city", "hangzhou" + random.nextInt());
            jsonObject.put("number", random.nextInt());
            ctx.collect(jsonObject.toJSONString());
            Thread.sleep(1000);
        }
    }

    @Override
    public void cancel() {
        isRunning = false;
    }
}
```

3.7.3 Obtaining Temporary Credentials from a Flink Job's Agency for Accessing Other Cloud Services

Function

DLI provides a common interface to obtain temporary credentials from Flink job's agencies set by users during job launch. The interface encapsulates temporary credentials obtained from the job agency in the **com.huaweicloud.sdk.core.auth.BasicCredentials** class.

- Encapsulate temporary credentials obtained from the agency in the return value of **getCredentials()** of the **com.huaweicloud.sdk.core.auth.ICredentialProvider** interface.
- The return type is **com.huaweicloud.sdk.core.auth.BasicCredentials**.

- Only AKs, SKs, and security tokens can be obtained.
- After obtaining the AK, SK, and security token, query temporary credentials by referring to **Using CSMS to Change Hard-coded Database Account Passwords**.

Notes and Constraints

- Agency authorization for accessing temporary credentials is only supported in Flink 1.15.
  - When creating a Flink job, select version 1.15.
  - The information of the agency that allows DLI to access DEW has been configured for the job. **flink.dli.job.agency.name** indicates the custom agency name.

    For details about how to create a custom agency, see **Customizing DLI Agency Permissions**.

    Note that double quotes ("") or single quotes ('') are not required when configuring parameters.
- The Flink 1.15 basic image has built-in **huaweicloud-sdk-core** 3.1.62.

Preparing the Environment

Development tools such as IntelliJ IDEA and other development tools, JDK, and Maven have been installed and configured.

Dependency package in POM file configurations

```
<dependency>
   <groupId>com.huaweicloud.sdk</groupId>
   <artifactId>huaweicloud-sdk-core</artifactId>
   <version>3.1.62</version>
   <scope>provided</scope>
</dependency>
```

Sample Code

This section's Java sample code demonstrates how to obtain BasicCredentials and retrieve a temporary agency's AK, SK, and security token.

- **Obtaining job agency credentials using Flink UDFs**

```
package com.huawei.dli.demo;

import static com.huawei.dli.demo.utils.DLIJobAgencyCredentialUtils.getICredentialProvider;

import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.core.auth.ICredentialProvider;

import org.apache.flink.table.functions.FunctionContext;
import org.apache.flink.table.functions.ScalarFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class GetUserAgencyCredentialUDF extends ScalarFunction {
   private static final Logger LOG = LoggerFactory.getLogger(GetUserAgencyCredentialUDF.class);

   ICredentialProvider credentialProvider;

   @Override
   public void open(FunctionContext context) throws Exception {
```



```
        credentialProvider = getICredentialProvider();
    }

    public String eval(String value) {
        BasicCredentials basicCredentials = (BasicCredentials) credentialProvider.getCredentials();

        String ak = basicCredentials.getAk();
        String sk = basicCredentials.getSk();
        String securityToken = basicCredentials.getSecurityToken();
        LOG.info(">>> ak " + ak + " sk " + sk.length() + " token " + securityToken.length());
        return value + "_demo";
    }
}
```

- **Obtaining job agency credentials for Flink Jar jobs**

```
package com.huawei.dli.demo;

import static com.huawei.dli.demo.utils.DLIJobAgencyCredentialUtils.getICredentialProvider;

import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.core.auth.ICredentialProvider;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class GetUserCredentialsFlinkStream {
    private static final Logger LOG = LoggerFactory.getLogger(GetUserCredentialsFlinkStream.class);

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<String> stream = streamEnv.addSource(new DataGen()).disableChaining();
        ICredentialProvider credentialProvider = getICredentialProvider();

        BasicCredentials basicCredentials = (BasicCredentials) credentialProvider.getCredentials();
        String ak = basicCredentials.getAk();
        String sk = basicCredentials.getSk();
        String securityToken = basicCredentials.getSecurityToken();
        LOG.info(">>" + " ak " + ak + " sk " + sk.length() + " token " + securityToken.length());

        stream.print();
        streamEnv.execute("GetUserCredentialsFlinkStream");
    }
}
```

- **Tool class for obtaining job agencies**

```
package com.huawei.dli.demo.utils;

import com.huaweicloud.sdk.core.auth.ICredentialProvider;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.ArrayList;
import java.util.List;
import java.util.ServiceLoader;

public class DLIJobAgencyCredentialUtils {

    public static ICredentialProvider getICredentialProvider() {
        List<ICredentialProvider> credentialProviders = new ArrayList<>();
        ServiceLoader.load(ICredentialProvider.class, StreamExecutionEnvironment.class.getClassLoader())
            .iterator()
            .forEachRemaining(credentialProviders::add);

        if (credentialProviders.size() != 1) {
            throw new RuntimeException("Failed to obtain temporary user credential");
        }
```



```
        return credentialProviders.get(0);
    }
}
```


4 Spark Jar Jobs

4.1 Using Spark Jar Jobs to Read and Query OBS Data

Scenario

DLI is fully compatible with open-source **Apache Spark** and allows you to import, query, analyze, and process job data by programming. This section describes how to write a Spark program to read and query OBS data, compile and package the code, and submit it to a Spark Jar job.

Environment Preparations

Before you start, set up the development environment.

Table 4-1 Spark Jar job development environment

Item	Description
OS	Windows 7 or later
JDK	JDK 1.8.
IntelliJ IDEA	This tool is used for application development. The version of the tool must be 2019.1 or other compatible versions.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.

Development Process

The following figure shows the process of developing a Spark Jar job.

Figure 4-1 Development process

Table 4-2 Process description

No.	Phase	Software Portal	Description
1	Create a queue for general use.	DLI console	The DLI queue is created for running your job.
2	Upload data to an OBS bucket.	OBS console	The test data needs to be uploaded to your OBS bucket.
3	Create a Maven project and configure the POM file.	IntelliJ IDEA	Write your code by referring to the sample code for reading data from OBS.
4	Write code.		
5	Debug, compile, and pack the code into a Jar package.		
6	Upload the Jar package to OBS and DLI.	OBS console DLI console	You can upload the generated Spark JAR package to an OBS directory and DLI program package.
7	Create a Spark Jar Job.	DLI console	The Spark Jar job is created and submitted on the DLI console.
8	Check the job execution result.	DLI console	You can check the job status and run logs.

Step 1: Create a Queue for General Purpose

Create a queue before submitting Spark jobs. In this example, we will create a general-purpose queue named **sparktest**.

1. Log in to the DLI management console.
2. In the navigation pane on the left, choose **Resources** > **Resource Pool**.
3. On the displayed page, click **Buy Resource Pool** in the upper right corner.
4. On the displayed page, set the parameters.

   In this example, we will buy the resource pool in the **CN East-Shanghai2** region. **Table 4-3** describes the parameters.

Table 4-3 Parameters

Parameter	Description	Example Value
Region	Select a region where you want to buy the elastic resource pool.	CN East-Shanghai2
Project	Project uniquely preset by the system for each region	Default
Name	Name of the elastic resource pool	dli_resource_pool
Specifications	Specifications of the elastic resource pool	Standard
CU Range	The maximum and minimum CUs allowed for the elastic resource pool	64-64
CIDR Block	CIDR block the elastic resource pool belongs to. If you use an enhanced datasource connection, this CIDR block cannot overlap that of the data source. **Once set, this CIDR block cannot be changed.**	172.16.0.0/19
Enterprise Project	Select an enterprise project for the elastic resource pool.	default

5. Click **Buy**.
6. Click **Submit**.
7. In the elastic resource pool list, locate the pool you just created and click **Add Queue** in the **Operation** column.
8. Set the basic parameters listed below.

Table 4-4 Basic parameters for adding a queue

Parameter	Description	Example Value
Name	Name of the queue to add	dli_queue_01
Type	Type of the queue • To execute SQL jobs, select **For SQL**. • To execute Flink or Spark jobs, select **For general purpose**.	_
Engine	SQL queue engine. The options are **Spark** and **HetuEngine**.	_
Enterprise Project	Select an enterprise project.	default

9. Click **Next** and configure scaling policies for the queue.

   Click **Create** to add a scaling policy with varying priority, period, minimum CUs, and maximum CUs.

   Figure 4-2 shows the scaling policy configured in this example.

   Figure 4-2 Configuring a scaling policy when adding a queue

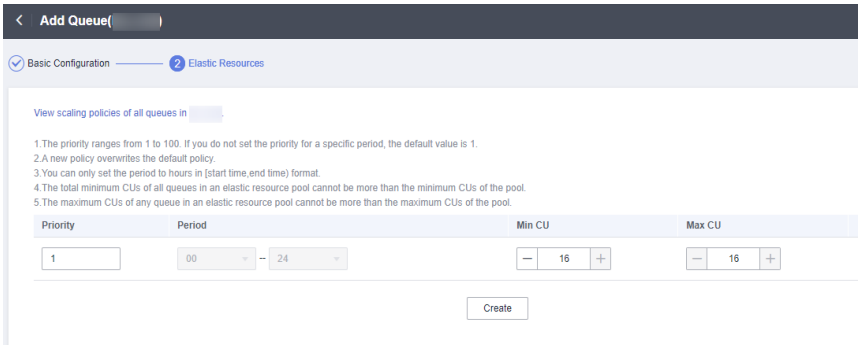


   Table 4-5 Scaling policy parameters

Parameter	Description	Example Value
Priority	Priority of the scaling policy in the current elastic resource pool. A larger value indicates a higher priority. In this example, only one scaling policy is configured, so its priority is set to **1** by default.	1
Period	The first scaling policy is the default policy, and its **Period** parameter configuration cannot be deleted or modified. The period for the scaling policy is from 00 to 24.	00–24
Min CU	Minimum number of CUs allowed by the scaling policy	16
Max CU	Maximum number of CUs allowed by the scaling policy	64

10. Click **OK**.

Step 2: Upload Data to OBS

1. Create the **people.json** file containing the following content:

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

2. Log in to the OBS console. On the **Buckets** page, click the name of the OBS bucket you created. In this example, the bucket name is **dli-test-obs01**.
3. On the **Objects** tab, click **Upload Object**. In the displayed dialog box, upload the **people.json** file to the root directory of the OBS bucket.

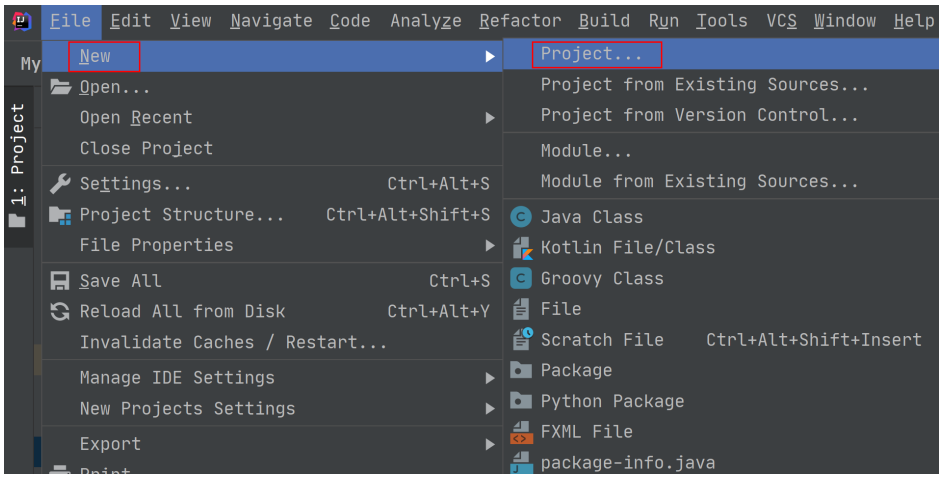
4. In the root directory of the OBS bucket, click **Create Folder** to create a folder and name it **result**.
5. Click the **result** folder, click **Create Folder** on the displayed page to create a folder and name it **parquet**.

Step 3: Create a Maven Project and Configure the pom Dependency

This step uses IntelliJ IDEA 2020.2 as an example.

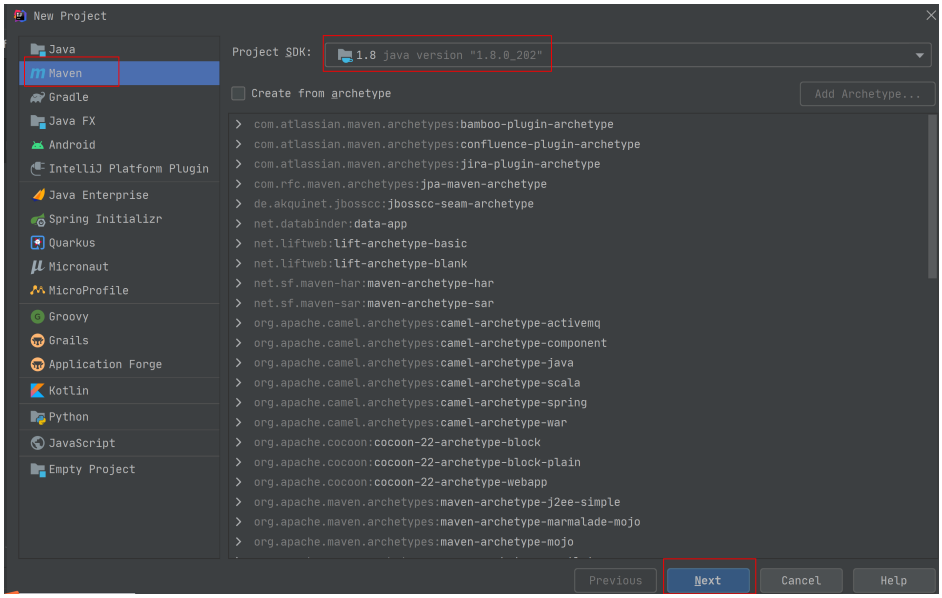
1. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

Figure 4-3 Creating a project



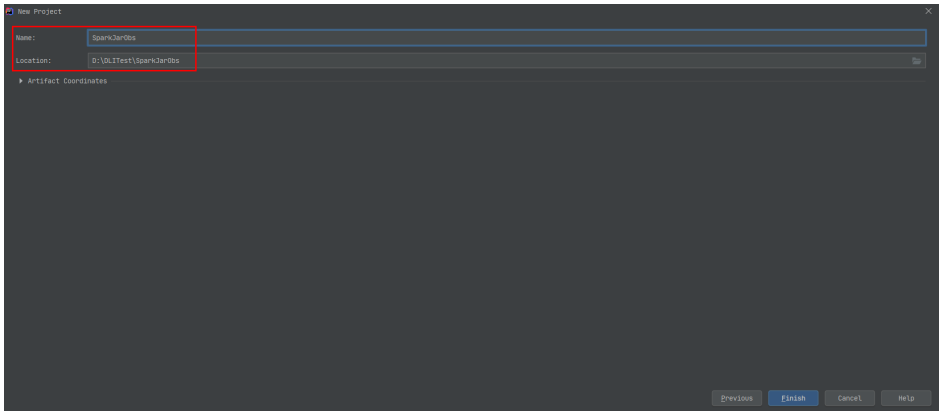
2. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

Figure 4-4 Creating a project



3. Set the project name, configure the storage path, and click **Finish**.

Figure 4-5 Creating a project

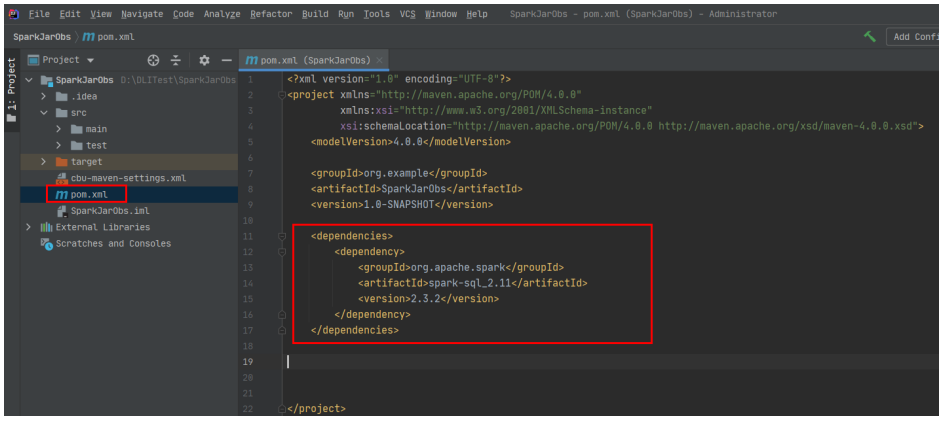


In this example, the Maven project name is **SparkJarObs**, and the project storage path is **D:\DLITest\SparkJarObs**.

4. Add the following content to the **pom.xml** file.

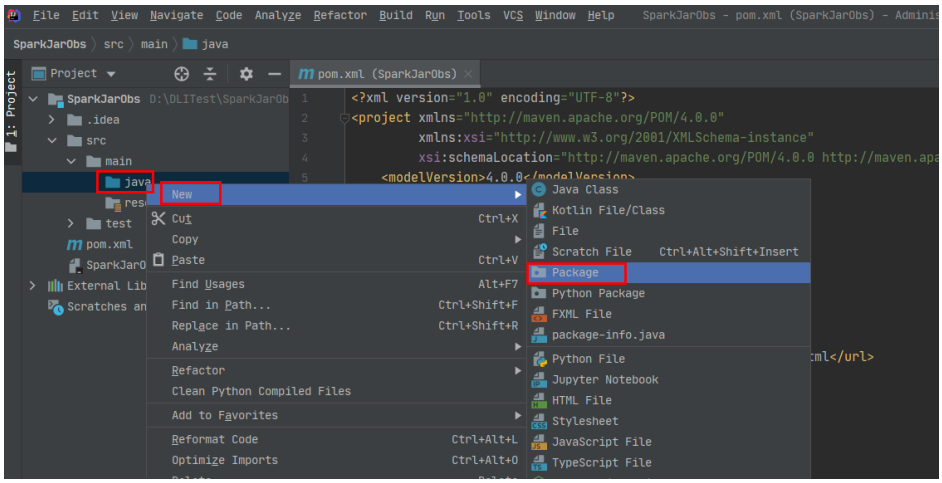
```
<dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
        </dependency>
</dependencies>
```

Figure 4-6 Modifying the **pom.xml** file



5. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

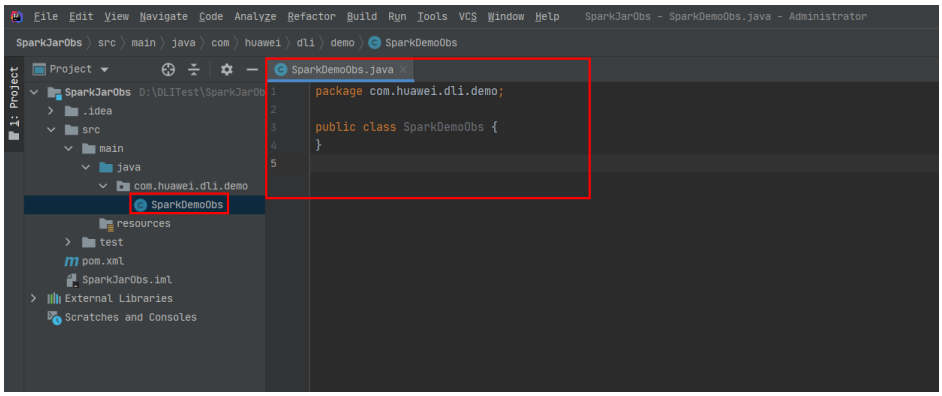
Figure 4-7 Creating a package



Set the package name as you need. In this example, set **Package** to **com.huawei.dli.demo**. Then, press **Enter**.

Create a Java Class file in the package path. In this example, the Java Class file is **SparkDemoObs**.

Figure 4-8 Creating a Java class file



Step 4: Write Code

Code the **SparkDemoObs** program to read the **people.json** file from the OBS bucket, create the temporary table **people**, and query data.

For the sample code, see **Sample Code**.

1. Import dependencies.

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.SparkSession;

import static org.apache.spark.sql.functions.col;
```

2. Create Spark session **spark** using the AK and SK of the current account.

```
SparkSession spark = SparkSession
        .builder()
        .config("spark.hadoop.fs.obs.access.key", "xxx")
        .config("spark.hadoop.fs.obs.secret.key", "yyy")
```



```
            .appName("java_spark_demo")
            .getOrCreate();
```

- Replace *xxx* of "spark.hadoop.fs.obs.access.key" with the AK of the account.
- Replace *yyy* of "spark.hadoop.fs.obs.secret.key" with the SK of the account.

For details about how to obtain the AK and SK, see **How Do I Obtain the AK/SK Pair?**

3. Read the **people.json** file from the OBS bucket.

   dli-test-obs01 is the name of the sample OBS bucket. Replace it with the actual OBS bucket name.

```
Dataset<Row> df = spark.read().json("obs://dli-test-obs01/people.json");
df.printSchema();
```

4. Create temporary table **people** to read data.

```
df.createOrReplaceTempView("people");
```

5. Query data in the **people** table.

```
Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
```

6. Export **people** table data in Parquet format to the **result/parquet** directory of the OBS bucket.

```
sqlDF.write().mode(SaveMode.Overwrite).parquet("obs://dli-test-obs01/result/parquet");
spark.read().parquet("obs://dli-test-obs01/result/parquet").show();
```

7. Disable the **spark** session.

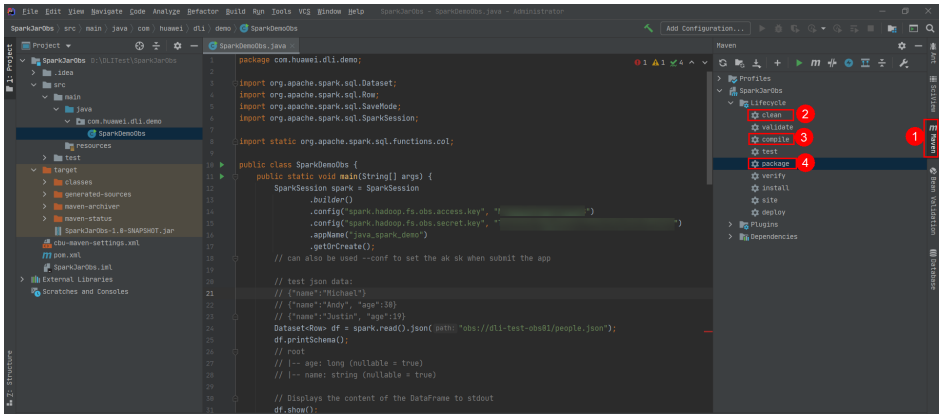
```
spark.stop();
```

Step 5: Debug, compile, and pack the code into a JAR package.

1. Double-click **Maven** in the tool bar on the right, and double-click **clean** and **compile** to compile the code.

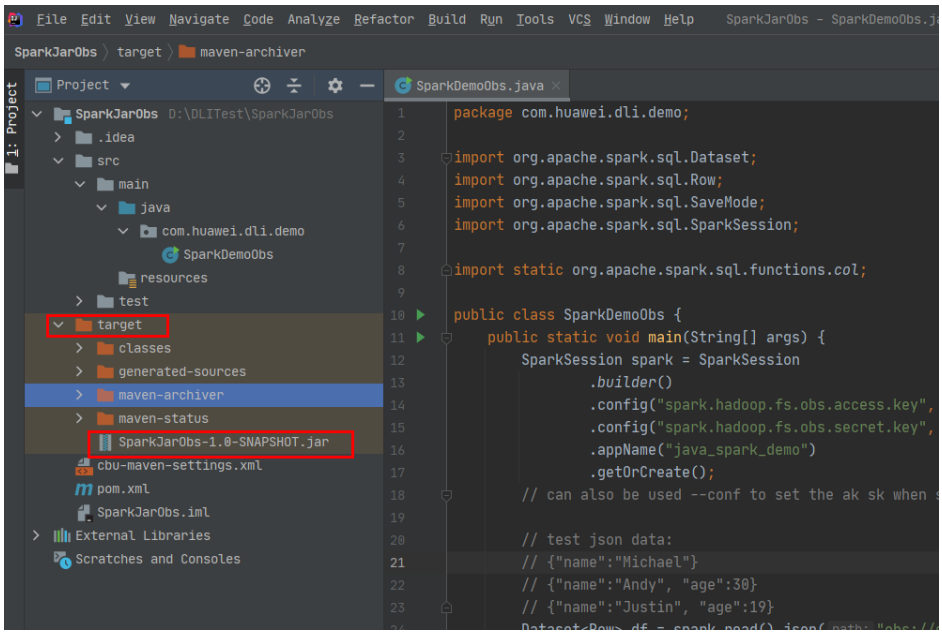
   After the compilation is successful, double-click **package**.

   Figure 4-9 Compiling and packaging



   The generated JAR package is stored in the **target** directory. In this example, **SparkJarObs-1.0-SNAPSHOT.jar** is stored in **D:\DLITest\SparkJarObs\target**.

Figure 4-10 Exporting the JAR file



Step 6: Upload the JAR Package to OBS and DLI

- **Spark 3.3 or later:**

  You can only set the **Application** parameter when creating a Spark job and select the required JAR file from OBS.

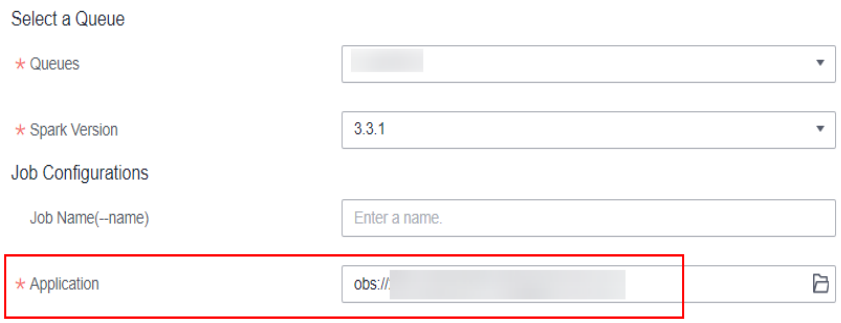
  a. Log in to the OBS console and upload the JAR file to the OBS path.

  b. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Spark Jobs**.

  c. Locate the row containing a desired job and click **Edit** in the **Operation** column.

  d. Set **Application** to the OBS path in **a**.

  Figure 4-11 Configuring the application

- **Versions earlier than Spark 3.3:**

  Upload the JAR file to OBS and DLI.

  a. Log in to the OBS console and upload the JAR file to the OBS path.

  b. Upload the file to DLI for package management.

i. Log in to the DLI management console and choose **Data Management** > **Package Management**.

ii. On the **Package Management** page, click **Create** in the upper right corner.

iii. In the **Create Package** dialog, set the following parameters:

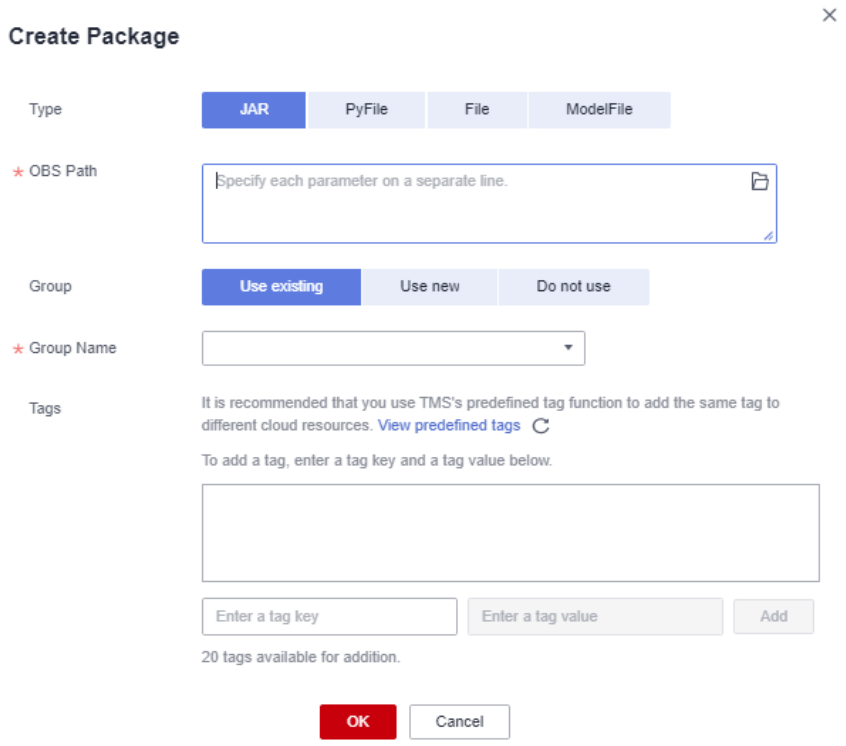
1) **Type**: Select **JAR**.

2) **OBS Path**: Specify the OBS path for storing the package.

3) Set **Group** and **Group Name** as required for package identification and management.

iv. Click **OK**.

Figure 4-12 Creating a package



Step 7: Create a Spark Jar Job

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Spark Jobs**.
2. On the **Spark Jobs** page, click **Create Job**.
3. On the displayed page, configure the following parameters:
   - **Queue**: Select the created queue. For example, select the queue **sparktest** created in **Step 1: Create a Queue for General Purpose**.
   - Select a supported Spark version from the drop-down list. The latest version is recommended.
   - **Job Name (--name)**: Name of the Spark Jar job. For example, **SparkTestObs**.

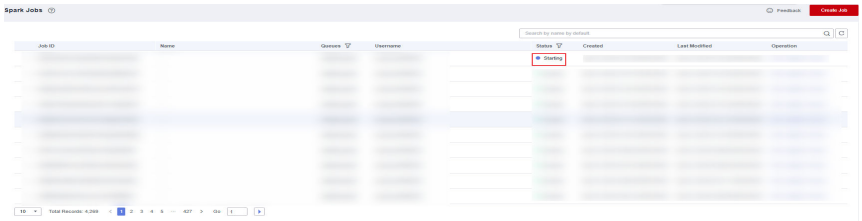
- **Application**: Select the package uploaded in **Step 6: Upload the JAR Package to OBS and DLI**. For example, select **SparkJarObs-1.0-SNAPSHOT.jar**.
- **Main Class (--class)**: The format is program package name + class name. For example, **com.huawei.dli.demo.SparkDemoObs**.

You do not need to set other parameters.

For more information about Spark JAR job submission, see **Creating a Spark Job**.

4. Click **Execute** to submit the Spark Jar job. On the **Spark Jobs** page, check the status of the job you submitted.

Figure 4-13 Job status



Step 8: Check Job Execution Result

1. On the **Spark Jobs** page, check the status of the job you submitted. The initial status is **Starting**.
2. If the job is successfully executed, the job status is **Finished**. Click **More** in the **Operation** column and select **Driver Logs** to check the run log.

Figure 4-14 Selecting Diver Logs

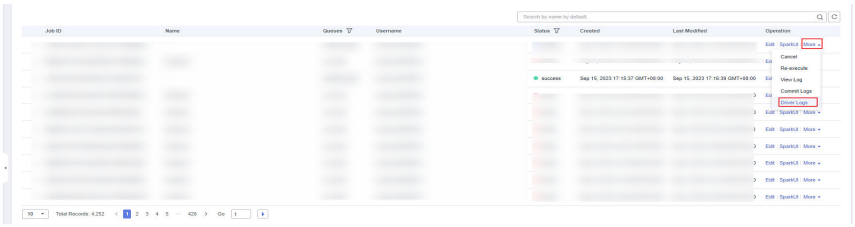
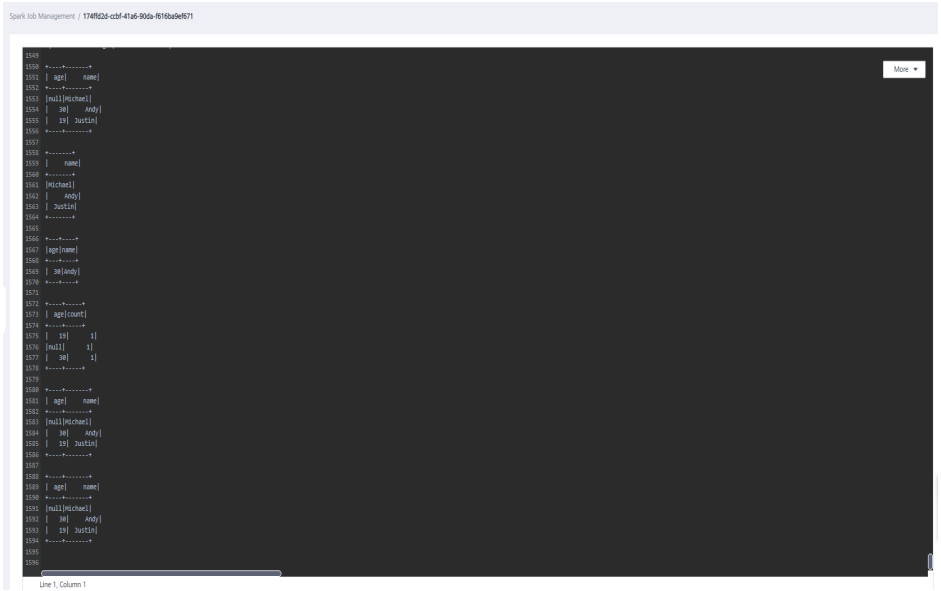


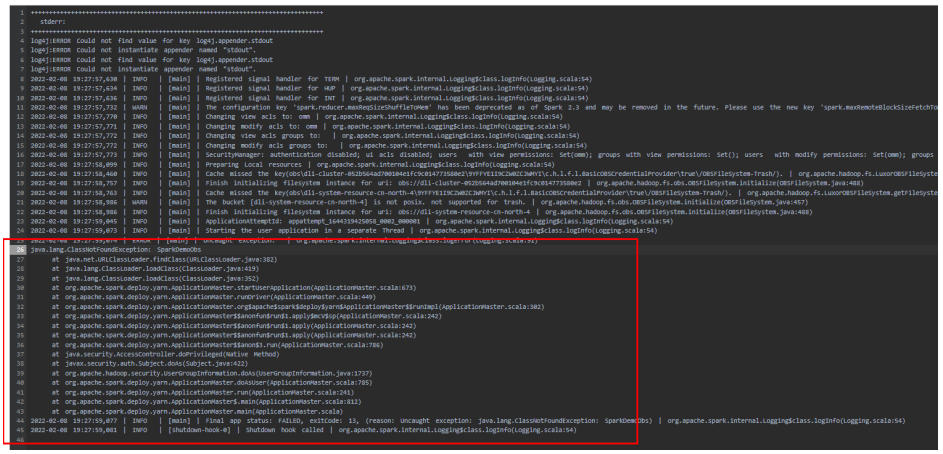
Figure 4-15 Driver logs



3. If the job is successfully executed, go to the **result/parquet** directory in the OBS bucket to check the generated **parquet** file.
4. If the job fails to be executed, click **More** in the **Operation** column and select **Driver Logs** to check detailed error information.

   For example, the following figure shows that when you create the Spark Jar job, you did not add the package path to the main class name.

Figure 4-16 Error information



In the **Operation** column, click **Edit**, change the value of **Main Class** to **com.huawei.dli.demo.SparkDemoObs**, and click **Execute** to run the job again.

Follow-up Guide

- If you want to use Spark Jar jobs to access other data sources, see **Using Spark Jobs to Access Data Sources of Datasource Connections**.
- If you want to create a database and table using a Spark Jar job, see **Using Spark Jobs to Access DLI Metadata**.

Sample Code

NOTE

Hard-coded or plaintext **access.key** and **secret.key** pose significant security risks. To ensure security, encrypt your AK and SK, store them in configuration files or environment variables, and decrypt them when needed.

```
package com.huawei.dli.demo;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.SparkSession;

import static org.apache.spark.sql.functions.col;

public class SparkDemoObs {
    public static void main(String[] args) {
        SparkSession spark = SparkSession
                .builder()
                .config("spark.hadoop.fs.obs.access.key", "xxx")
                .config("spark.hadoop.fs.obs.secret.key", "yyy")
                .appName("java_spark_demo")
                .getOrCreate();
        // can also be used --conf to set the ak sk when submit the app

        // test json data:
        // {"name":"Michael"}
        // {"name":"Andy", "age":30}
        // {"name":"Justin", "age":19}
        Dataset<Row> df = spark.read().json("obs://dli-test-obs01/people.json");
        df.printSchema();
        // root
        // |-- age: long (nullable = true)
        // |-- name: string (nullable = true)

        // Displays the content of the DataFrame to stdout
        df.show();
        // +----+-------+
        // | age|   name|
        // +----+-------+
        // |null|Michael|
        // |  30|   Andy|
        // |  19| Justin|
        // +----+-------+

        // Select only the "name" column
        df.select("name").show();
        // +-------+
        // |   name|
        // +-------+
        // |Michael|
        // |   Andy|
        // | Justin|
        // +-------+

        // Select people older than 21
        df.filter(col("age").gt(21)).show();
        // +---+----+
        // |age|name|
        // +---+----+
        // | 30|Andy|
        // +---+----+

        // Count people by age
        df.groupBy("age").count().show();
        // +----+-----+
        // | age|count|
```



```
        // +----+-----+
        // |  19|    1|
        // |null|    1|
        // |  30|    1|
        // +----+-----+

        // Register the DataFrame as a SQL temporary view
        df.createOrReplaceTempView("people");

        Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
        sqlDF.show();
        // +----+-------+
        // | age|   name|
        // +----+-------+
        // |null|Michael|
        // |  30|   Andy|
        // |  19| Justin|
        // +----+-------+

        sqlDF.write().mode(SaveMode.Overwrite).parquet("obs://dli-test-obs01/result/parquet");
        spark.read().parquet("obs://dli-test-obs01/result/parquet").show();

        spark.stop();
    }
}
```

4.2 Using the Spark Job to Access DLI Metadata

Scenario

DLI allows you to develop a program to create Spark jobs for operations related to databases, DLI or OBS tables, and table data. This example demonstrates how to develop a job by writing a Java program, and use a Spark job to create a database and table and insert table data.

NOTE

This feature is currently in beta testing. If you want to use it, submit a service ticket to apply for permission to access DLI metadata using Spark jobs.

Constraints

- You must create a queue to use Spark 3.1 for metadata access.
- The following cases are not supported:
  - If you create a database with a SQL job, you cannot write a program to create tables in that database.

    For example, the **testdb** database is created using the SQL editor of DLI. A program package for creating the **testTable** table in the **testdb** database does not work after it is submitted to a Spark Jar job.
- The following cases are supported:
  - You can create databases and tables in a SQL job, and read and insert data using SQL statements or a Spark program.
  - You can create databases and tables in a Spark job, and read and insert data using SQL statements or a Spark program.

Environment Preparations

Before developing a Spark job to access DLI metadata, set up a development environment that meets the following requirements.

Table 4-6 Development environment

Item	Description
OS	Windows 7 or later
JDK	JDK 1.8.
IntelliJ IDEA	This tool is used for application development. The version of the tool must be 2019.1 or other compatible versions.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.

Development Process

The following figure shows the process for developing a Spark job to access DLI metadata.

Figure 4-17 Development process

Table 4-7 Process description

No.	Phase	Software Portal	Description
1	Create a queue for general use.	DLI console	The DLI queue is created for running your job.
2	Configure the OBS file.	OBS console	• To create an OBS table, you need to upload the file to the OBS bucket. • Configure the path for storing DLI metadata. This folder is used to store DLI metadata in **spark.sql.warehouse.dir**.
3	Create a Maven project and configure the POM file.	IntelliJ IDEA	Write a program to create a DLI or OBS table by referring to the sample code.
4	Write code.		

No.	Phase	Software Portal	Description
5	Debug, compile, and pack the code into a Jar package.		
6	Upload the Jar package to OBS and DLI.	OBS console DLI console	You can upload the generated Spark Jar package to an OBS directory and DLI program package.
7	Create a Spark JAR job.	DLI console	The Spark Jar job is created and submitted on the DLI console.
8	Check execution result of the job.	DLI console	You can check the job status and run logs.

Step 1: Create a Queue for General Purpose

Create a queue before submitting Spark jobs. In this example, we will create a general-purpose queue named **sparktest**.

1. Log in to the DLI management console.
2. In the navigation pane on the left, choose **Resources** > **Resource Pool**.
3. On the displayed page, click **Buy Resource Pool** in the upper right corner.
4. On the displayed page, set the parameters.

   In this example, we will buy the resource pool in the **CN East-Shanghai2** region. **Table 4-8** describes the parameters.

Table 4-8 Parameters

Parameter	Description	Example Value
Region	Select a region where you want to buy the elastic resource pool.	CN East-Shanghai2
Project	Project uniquely preset by the system for each region	Default
Name	Name of the elastic resource pool	dli_resource_pool
Specifications	Specifications of the elastic resource pool	Standard

Parameter	Description	Example Value
CU Range	The maximum and minimum CUs allowed for the elastic resource pool	64-64
CIDR Block	CIDR block the elastic resource pool belongs to. If you use an enhanced datasource connection, this CIDR block cannot overlap that of the data source. **Once set, this CIDR block cannot be changed.**	172.16.0.0/19
Enterprise Project	Select an enterprise project for the elastic resource pool.	default

5. Click **Buy**.
6. Click **Submit**.
7. In the elastic resource pool list, locate the pool you just created and click **Add Queue** in the **Operation** column.
8. Set the basic parameters listed below.

   Table 4-9 Basic parameters for adding a queue

Parameter	Description	Example Value
Name	Name of the queue to add	dli_queue_01
Type	Type of the queue • To execute SQL jobs, select **For SQL**. • To execute Flink or Spark jobs, select **For general purpose**.	_
Engine	SQL queue engine. The options are **Spark** and **HetuEngine**.	_
Enterprise Project	Select an enterprise project.	default

9. Click **Next** and configure scaling policies for the queue.

   Click **Create** to add a scaling policy with varying priority, period, minimum CUs, and maximum CUs.

   Figure 4-18 shows the scaling policy configured in this example.

Figure 4-18 Configuring a scaling policy when adding a queue

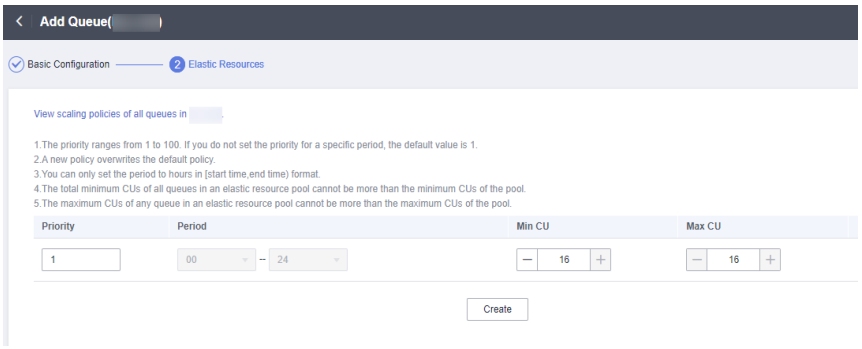


Table 4-10 Scaling policy parameters

Parameter	Description	Example Value
Priority	Priority of the scaling policy in the current elastic resource pool. A larger value indicates a higher priority. In this example, only one scaling policy is configured, so its priority is set to **1** by default.	1
Period	The first scaling policy is the default policy, and its **Period** parameter configuration cannot be deleted or modified. The period for the scaling policy is from 00 to 24.	00–24
Min CU	Minimum number of CUs allowed by the scaling policy	16
Max CU	Maximum number of CUs allowed by the scaling policy	64

10. Click **OK**.

Step 2: Configure the OBS Bucket File

1. To create an OBS table, upload data to the OBS bucket directory.

   Use the following sample data to create the **testdata.csv** file and upload it to an OBS bucket.

   ```
   12,Michael
   27,Andy
   30,Justin
   ```

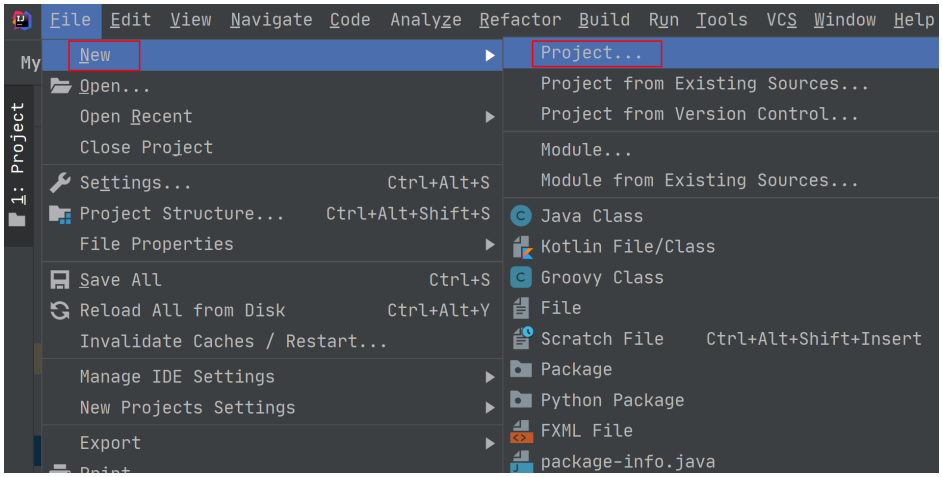
2. Log in to the OBS console. On the **Buckets** page, click the name of the OBS bucket you created. In this example, the bucket name is **dli-test-obs01**.
3. On the **Objects** tab, click **Upload Object**. In the displayed dialog box, upload the **testdata.csv** file to the root directory of the OBS bucket.
4. In the root directory of the OBS bucket, click **Create Folder** to create a folder and name it **warehousepath**. This folder is used to store DLI metadata in **spark.sql.warehouse.dir**.

Step 3: Create a Maven Project and Configure the POM Dependency

This step uses IntelliJ IDEA 2020.2 as an example.

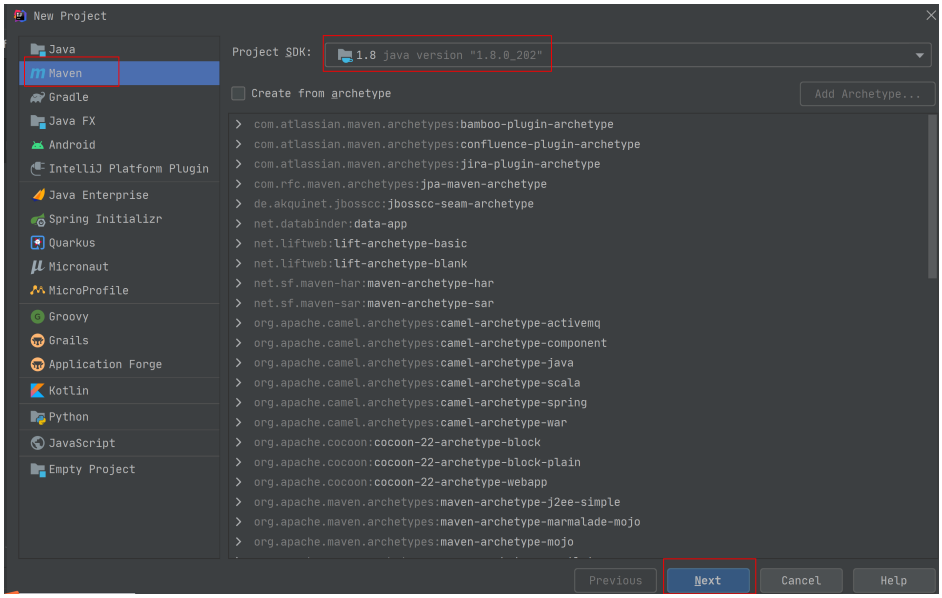
1. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

Figure 4-19 Creating a project



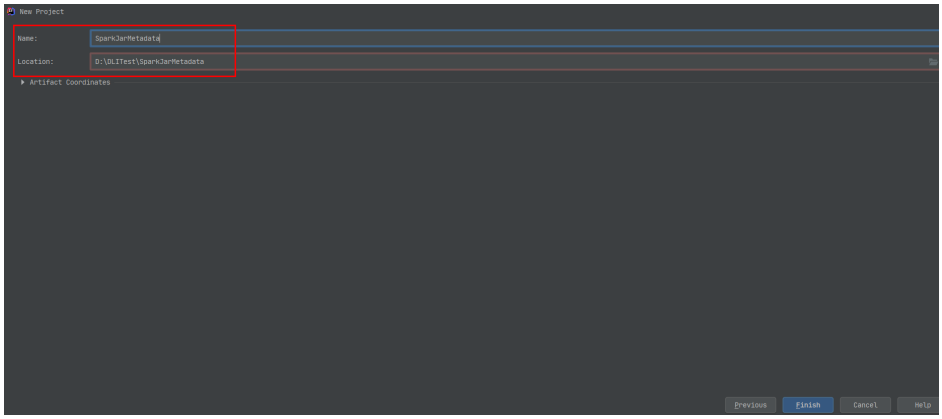
2. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

Figure 4-20 Selecting an SDK



3. Set the project name, configure the storage path, and click **Finish**.

Figure 4-21 Creating a project

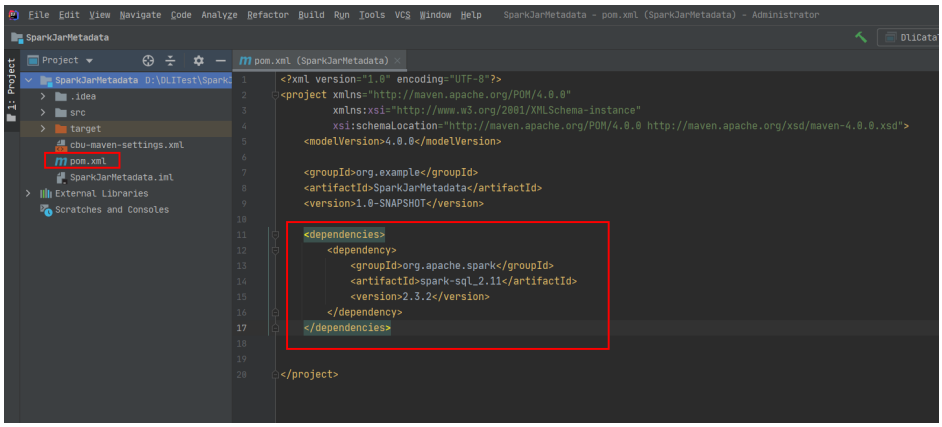


In this example, the Maven project name is **SparkJarMetadata**, and the project storage path is **D:\DLITest\SparkJarMetadata**.

4. Add the following content to the **pom.xml** file.

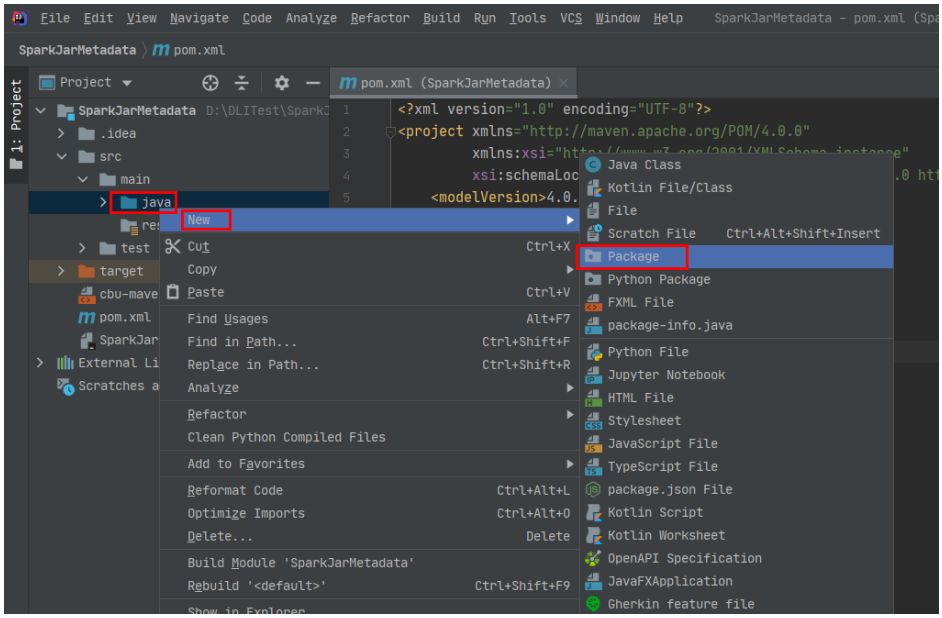
```
<dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
        </dependency>
</dependencies>
```

Figure 4-22 Modifying the **pom.xml** file



5. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

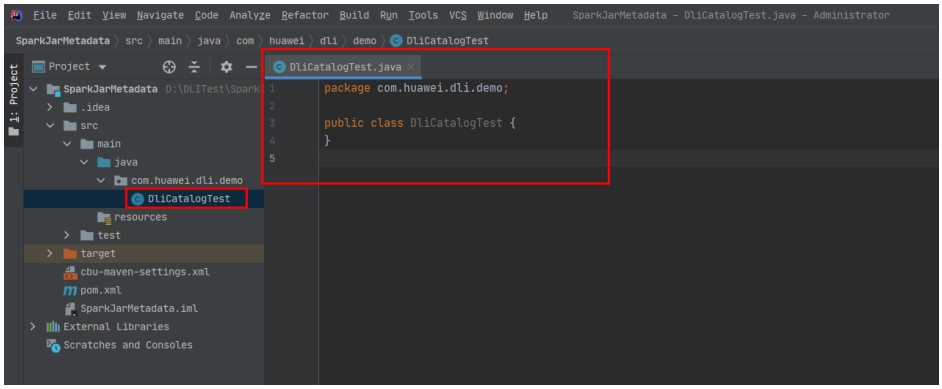
Figure 4-23 Creating a package



Set the package name as you need. In this example, set **Package** to **com.huawei.dli.demo** and press **Enter**.

Create a Java Class file in the package path. In this example, the Java Class file is **DliCatalogTest**.

Figure 4-24 Creating a Java class file



Step 4: Write Code

Write the DliCatalogTest program to create a database, DLI table, and OBS table.

For the sample code, see **Java Example Code**.

1. Import the dependency.
   ```
   import org.apache.spark.sql.SparkSession;
   ```
2. Create a SparkSession instance.

   When you create a SparkSession, you need to specify **spark.sql.session.state.builder**, **spark.sql.catalog.class**, and **spark.sql.extensions** parameters as configured in the following example.

- Spark 2.3.*x*

```
SparkSession spark = SparkSession
                .builder()
                .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
                .config("spark.sql.catalog.class",
"org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
                .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
                .appName("java_spark_demo")
                .getOrCreate();
```

- Spark 2.4.*x*

```
SparkSession spark = SparkSession
                .builder()
                .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
                .config("spark.sql.catalog.class",
"org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
                .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
                .config("spark.sql.hive.implementation","org.apache.spark.sql.hive.client.DliHiveClientI
mpl")
                .appName("java_spark_demo")
                .getOrCreate();
```

- Spark 3.1.*x*

```
SparkSession spark = SparkSession
                .builder()
                .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
                .config("spark.sql.catalog.class",
"org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
                .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
                .appName("java_spark_demo")
                .getOrCreate();
```

- Spark 3.3.*x*

```
SparkSession spark = SparkSession
        .builder()
        .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.DliLakeHouseBuilder")
        .config("spark.sql.catalog.class", "org.apache.spark.sql.hive.DliLakeHouseCatalog")
        .appName("java_spark_demo")
        .getOrCreate();
```

3. Create a database.

   The following sample code shows how to create a database and named **test_sparkapp**.

```
spark.sql("create database if not exists test_sparkapp").collect();
```

4. Create a DLI table and insert test data.

```
spark.sql("drop table if exists test_sparkapp.dli_testtable").collect();
spark.sql("create table test_sparkapp.dli_testtable(id INT, name STRING)").collect();
spark.sql("insert into test_sparkapp.dli_testtable VALUES (123,'jason')").collect();
spark.sql("insert into test_sparkapp.dli_testtable VALUES (456,'merry')").collect();
```

5. Create an OBS Table. Replace the OBS path in the following example with the path you set in **Step 2: Configure the OBS Bucket File**.

```
spark.sql("drop table if exists test_sparkapp.dli_testobstable").collect();
spark.sql("create table test_sparkapp.dli_testobstable(age INT, name STRING) using csv options (path
'obs://dli-test-obs01/testdata.csv')").collect();
```

6. Disable the **spark** session.

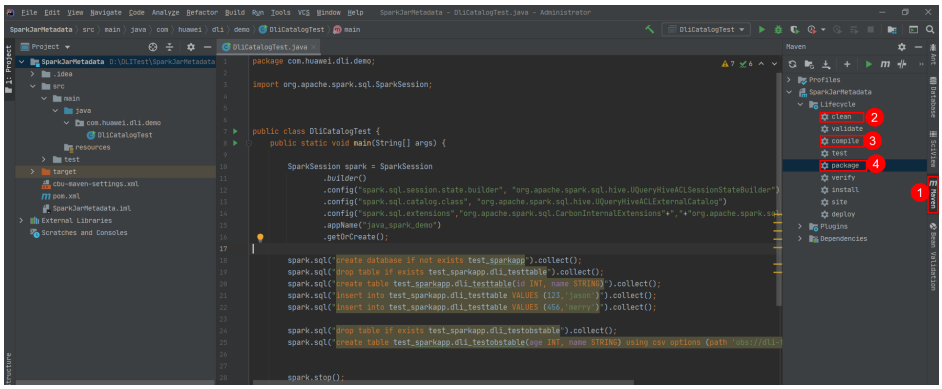
```
spark.stop();
```

Step 5: Debug, Compile, and Pack the Code into a Jar Package.

1. Double-click **Maven** in the tool bar on the right, and double-click **clean** and **compile** to compile the code.

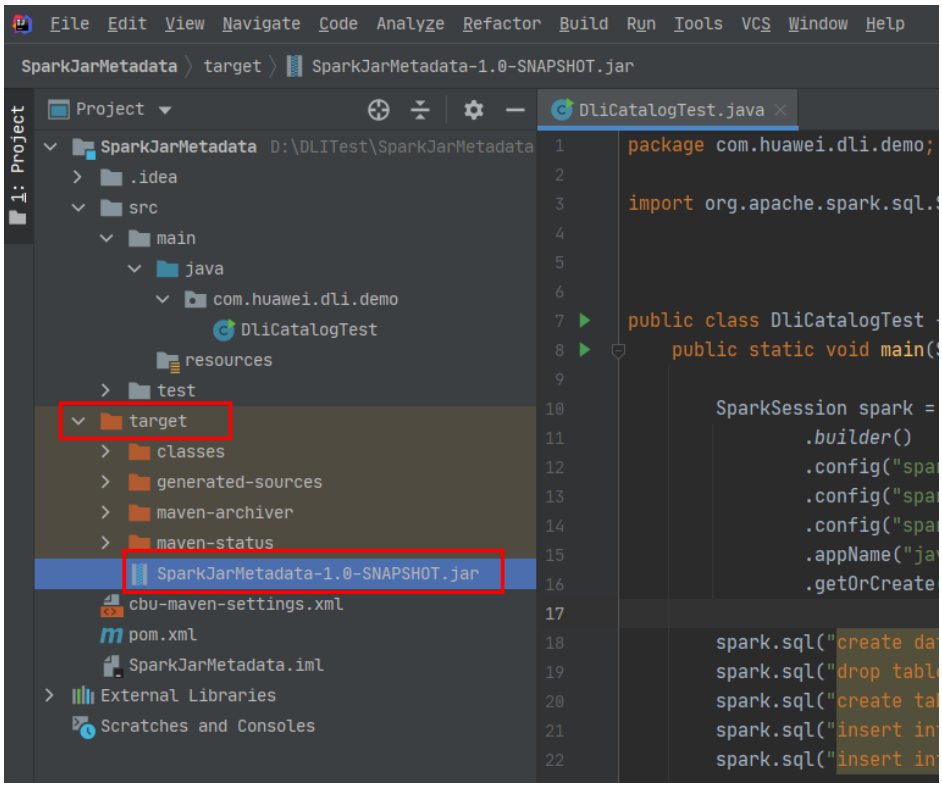
After the compilation is successful, double-click **package**.

Figure 4-25 Compiling and packaging



The generated JAR package is stored in the **target** directory. In this example, **SparkJarMetadata-1.0-SNAPSHOT.jar** is stored in **D:\DLITest\SparkJarMetadata\target**.

Figure 4-26 Exporting the JAR file



Step 6: Upload the JAR Package to OBS and DLI

- **Spark 3.3 or later:**

  You can only set the **Application** parameter when creating a Spark job and select the required JAR file from OBS.

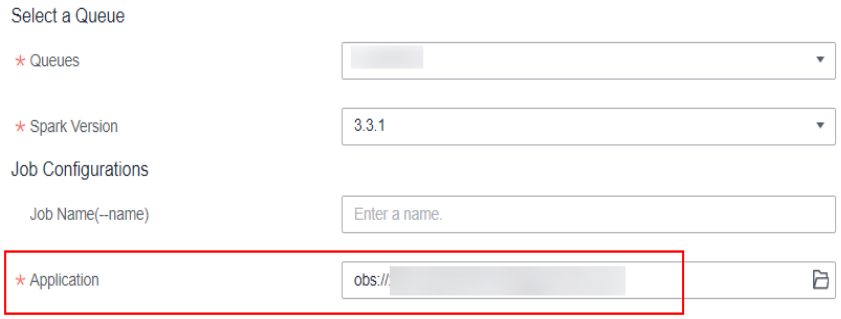
  a. Log in to the OBS console and upload the JAR file to the OBS path.

b. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Spark Jobs**.

c. Locate the row containing a desired job and click **Edit** in the **Operation** column.

d. Set **Application** to the OBS path in **a**.

Figure 4-27 Configuring the application



- **Versions earlier than Spark 3.3:**

  Upload the JAR file to OBS and DLI.

  a. Log in to the OBS console and upload the JAR file to the OBS path.

  b. Upload the file to DLI for package management.

     i. Log in to the DLI management console and choose **Data Management** > **Package Management**.

     ii. On the **Package Management** page, click **Create** in the upper right corner.

     iii. In the **Create Package** dialog, set the following parameters:

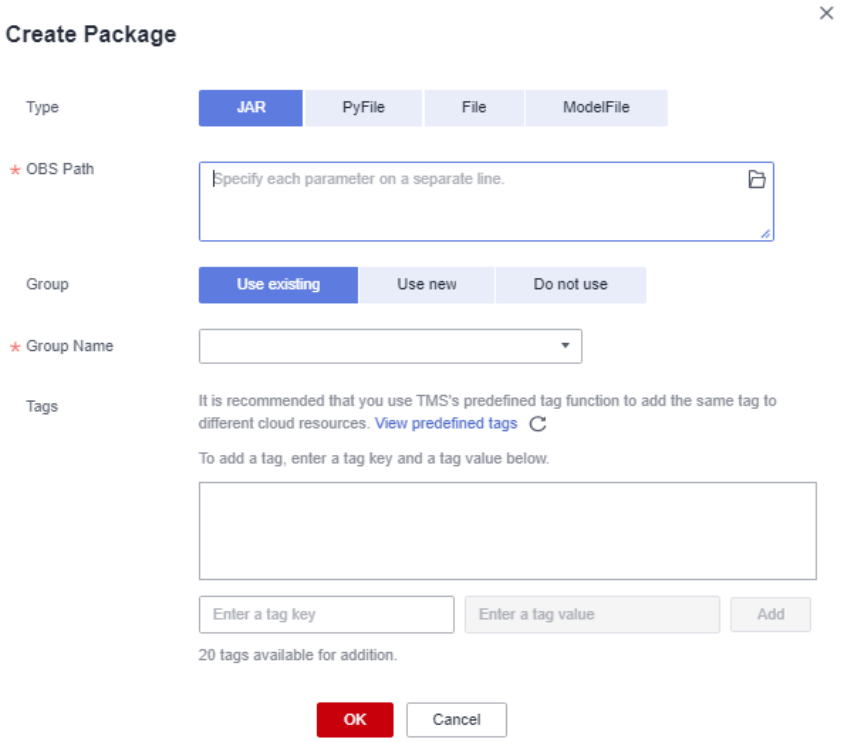
        1) **Type**: Select **JAR**.

        2) **OBS Path**: Specify the OBS path for storing the package.

        3) Set **Group** and **Group Name** as required for package identification and management.

     iv. Click **OK**.

Figure 4-28 Creating a package



Step 7: Create a Spark Jar Job

1. Log in to the DLI console. In the navigation pane, choose **Job Management** > **Spark Jobs**.
2. On the **Spark Jobs** page, click **Create Job** in the upper right corner.
3. On the job creation page, set the job parameters.

   Table 4-11 describes the parameters. Retain the default values for other parameters.

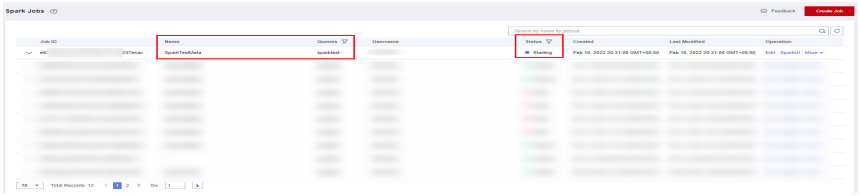
Table 4-11 Spark Jar job parameters

Parameter	Value
Queue	Select the DLI queue created for general purpose. For example, select the queue **sparktest** created in **Step 1: Create a Queue for General Purpose**.
Spark Version	Select a Spark version. Select a supported Spark version from the drop-down list. The latest version is recommended.
Job Name (--name)	Name of a custom Spark Jar job. For example, **SparkTestMeta**.
Application	Select the package uploaded to DLI in **Step 6: Upload the JAR Package to OBS and DLI**. For example, select **SparkJarMetadata-1.0-SNAPSHOT.jar**.

Parameter	Value
Main Class (--class)	The format is program package name + class name. For example, **com.huawei.dli.demo.DliCatalogTest**.
Spark Arguments (--conf)	spark.dli.metaAccess.enable=true spark.sql.warehouse.dir=*obs://dli-test-obs01/warehousepath* **NOTE** Set **spark.sql.warehouse.dir** to the OBS path that is specified in **Step 2: Configure the OBS Bucket File**.
Access Metadata	Select **Yes**.

4. Click **Execute** to submit the Spark Jar job. On the **Spark Jobs** page, check the status of the job you submitted.

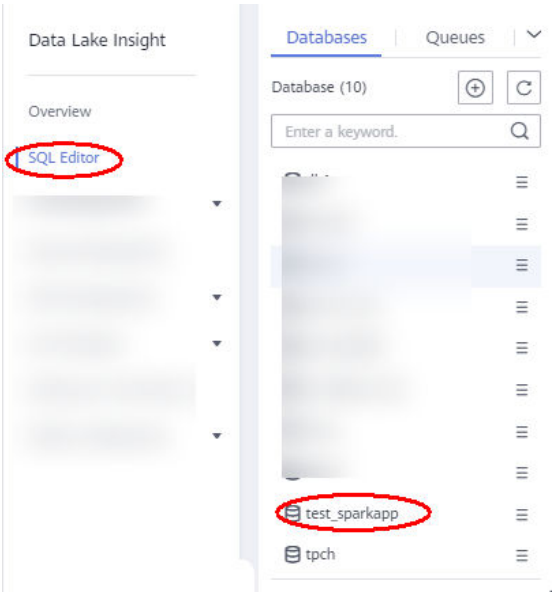
   Figure 4-29 Checking the job execution status



Step 8: Check Job Execution Result

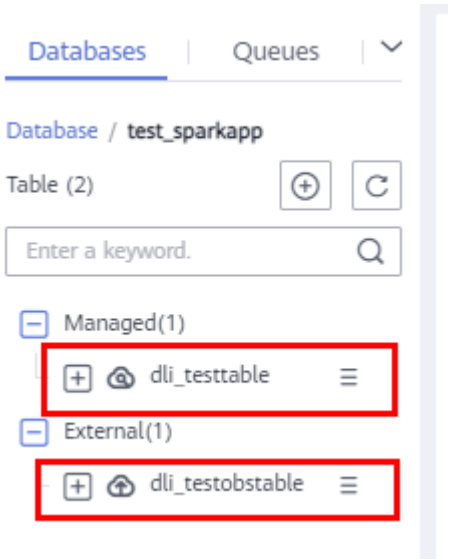
1. On the **Spark Jobs** page, check the status of the job you submitted. The initial status is **Starting**.
2. If the job is successfully executed, the job status is **Finished**. Perform the following operations to check the created database and table:
   a. On the DLI console, choose **SQL Editor** in the left navigation pane. The created database **test_sparkapp** is displayed in the database list.

Figure 4-30 Checking the created database



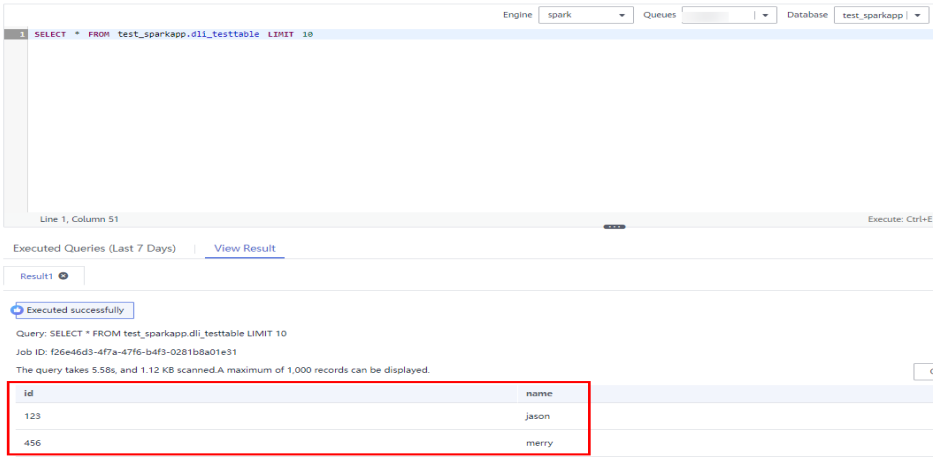
b. Double-click the database name to check the created DLI and OBS tables in the database.

Figure 4-31 Checking a table



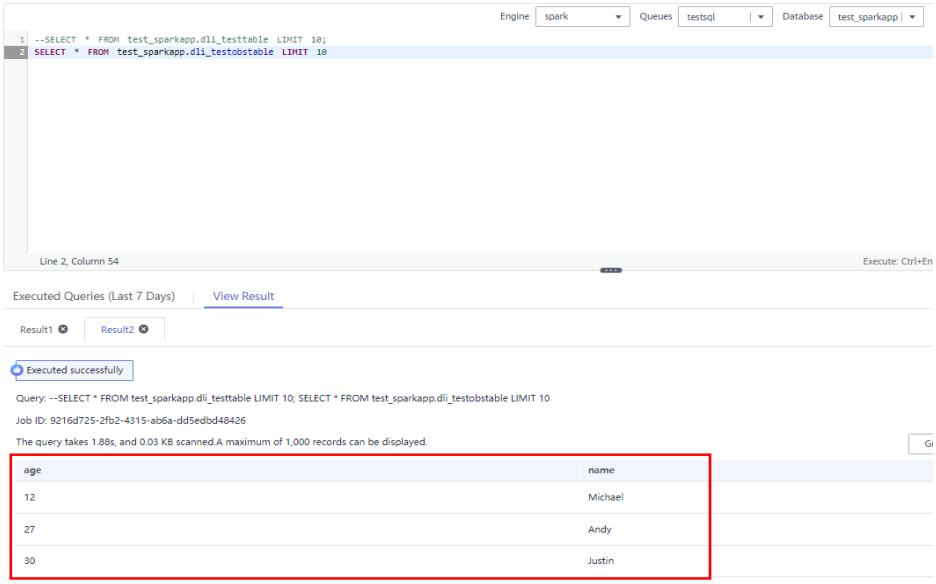
c. Double-click **dli_testtable** and click **Execute** to query data in the table.

Figure 4-32 Querying data in the table



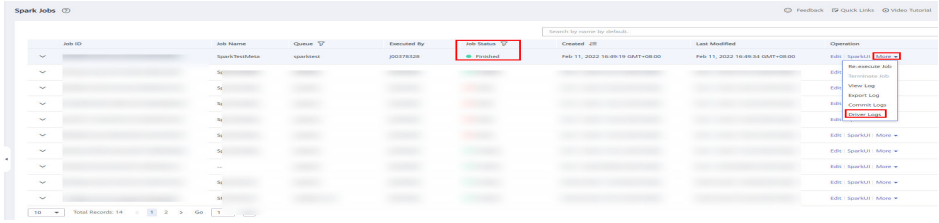
d. Comment out the statement for querying the DLI table, double-click the OBS table **dli_testobstable**, and click **Execute** to query the OBS table data.

Figure 4-33 Querying data in the OBS table



3. If the job is successfully executed, the job status is **Failed**. Click **More** in the **Operation** column and select **Driver Logs** to check the run log.

Figure 4-34 Checking logs



After the fault is rectified, click **Edit** in the **Operation** column of the job, modify job parameters, and click **Execute** to run the job again.

Follow-up Guide

- If you want to use Spark Jar jobs to access other data sources, see **Using Spark Jobs to Access Data Sources of Datasource Connections**.
- For details about the syntax for creating a DLI table, see **Creating a DLI Table**. For details about the syntax for creating an OBS table, see **Creating an OBS Table**.
- If you submit the job by calling an API, perform the following operations:

  Call the API for creating a batch processing job. The following table describes the request parameters.

  For details about API parameters, see **Creating a Batch Processing Job**.

  - Set **catalog_name** in the request to **dli**.
  - Add **"spark.dli.metaAccess.enable":"true"** to the CONF file.

    Configure **"spark.sql.warehouse.dir": "obs://bucket/warehousepath"** in the CONF file if you need to run the DDL.

    The following example provided you with the complete API request.

    ```
    {
       "queue":"citest",
       "file":"SparkJarMetadata-1.0-SNAPSHOT.jar",
       "className":"DliCatalogTest",
       "conf":{"spark.sql.warehouse.dir": "obs://bucket/warehousepath",
       "spark.dli.metaAccess.enable":"true"},
       "sc_type":"A",
       "executorCores":1,
       "numExecutors":6,
       "executorMemory":"4G",
       "driverCores":2,
       "driverMemory":"7G",
       "catalog_name": "dli"
    }
    ```

Java Example Code

This example uses Java for coding. The complete sample code is as follows:

```
package com.huawei.dli.demo;

import org.apache.spark.sql.SparkSession;

public class DliCatalogTest {
   public static void main(String[] args) {

      SparkSession spark = SparkSession
            .builder()
            .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
            .config("spark.sql.catalog.class", "org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
            .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
            .appName("java_spark_demo")
            .getOrCreate();

      spark.sql("create database if not exists test_sparkapp").collect();
      spark.sql("drop table if exists test_sparkapp.dli_testtable").collect();
      spark.sql("create table test_sparkapp.dli_testtable(id INT, name STRING)").collect();
      spark.sql("insert into test_sparkapp.dli_testtable VALUES (123,'jason')").collect();
      spark.sql("insert into test_sparkapp.dli_testtable VALUES (456,'merry')").collect();

      spark.sql("drop table if exists test_sparkapp.dli_testobstable").collect();
      spark.sql("create table test_sparkapp.dli_testobstable(age INT, name STRING) using csv options (path
'obs://dli-test-obs01/testdata.csv')").collect();
```



```
        spark.stop();

    }
}
```

Scala Example Code

```
object DliCatalogTest {
  def main(args:Array[String]): Unit = {
    val sql = args(0)
    val runDdl =
Try(args(1).toBoolean).getOrElse(true)
    System.out.println(s"sql is $sql
runDdl is $runDdl")
    val sparkConf = new SparkConf(true)
    sparkConf
      .set("spark.sql.session.state.builder","org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
      .set("spark.sql.catalog.class","org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
    sparkConf.setAppName("dlicatalogtester")

    val spark = SparkSession.builder
      .config(sparkConf)
      .enableHiveSupport()
      .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
      .appName("SparkTest")
      .getOrCreate()

    System.out.println("catalog is "
+ spark.sessionState.catalog.toString)
    if (runDdl) {
      val df = spark.sql(sql).collect()
    } else {
      spark.sql(sql).show()
    }

    spark.close()
  }

}
```

Example Python Code

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

from __future__ import print_function

import sys

from pyspark.sql import SparkSession

if __name__ == "__main__":
    url = sys.argv[1]
    creatTbl = "CREATE TABLE test_sparkapp.dli_rds USING JDBC OPTIONS ('url'='jdbc:mysql://%s'," \
               "'driver'='com.mysql.jdbc.Driver','dbtable'='test.test'," \
               " 'passwdauth' = 'DatasourceRDSTest_pwd','encryption' = 'true')" % url

    spark = SparkSession \
        .builder \
        .enableHiveSupport() \
.config("spark.sql.session.state.builder","org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
\
.config("spark.sql.catalog.class", "org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog") \
.config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension") \
        .appName("python Spark test catalog") \
        .getOrCreate()

    spark.sql("CREATE database if not exists test_sparkapp").collect()
```



```
spark.sql("drop table if exists test_sparkapp.dli_rds").collect()
spark.sql(creatTbl).collect()
spark.sql("select * from test_sparkapp.dli_rds").show()
spark.sql("insert into table test_sparkapp.dli_rds select 12,'aaa'").collect()
spark.sql("select * from test_sparkapp.dli_rds").show()
spark.sql("insert overwrite table test_sparkapp.dli_rds select 1111,'asasasa'").collect()
spark.sql("select * from test_sparkapp.dli_rds").show()
spark.sql("drop table test_sparkapp.dli_rds").collect()
spark.stop()
```

4.3 Using Spark Jobs to Access Data Sources of Datasource Connections

4.3.1 Overview

DLI supports the native Spark DataSource capability and other extended capabilities. You can use SQL statements or Spark jobs to access other data storage services and import, query, analyze, and process data. Currently, DLI supports the following datasource access services: CloudTable, Cloud Search Service (CSS), Distributed Cache Service (DCS), Document Database Service (DDS), GaussDB(DWS), MapReduce Service (MRS), and Relational Database Service (RDS). To use the datasource capability of DLI, you need to create a datasource connection first.

For details about operations on the management console, see **Data Lake Insight User Guide**.

When you use Spark jobs to access other data sources, you can use Scala, PySpark, or Java to program functions for the jobs.

- For details about CSS, see **What Is Cloud Search Service?**
- For details about DCS, see **What Is DCS?**
- For details about DDS, see **What Is DDS?**
- For details about GaussDB(DWS), see **What Is GaussDB(DWS)?**
- For details about MRS, see **What Is MRS?**
- For details about RDS, see **What Is RDS?**

4.3.2 Connecting to CSS

4.3.2.1 CSS Security Cluster Configuration

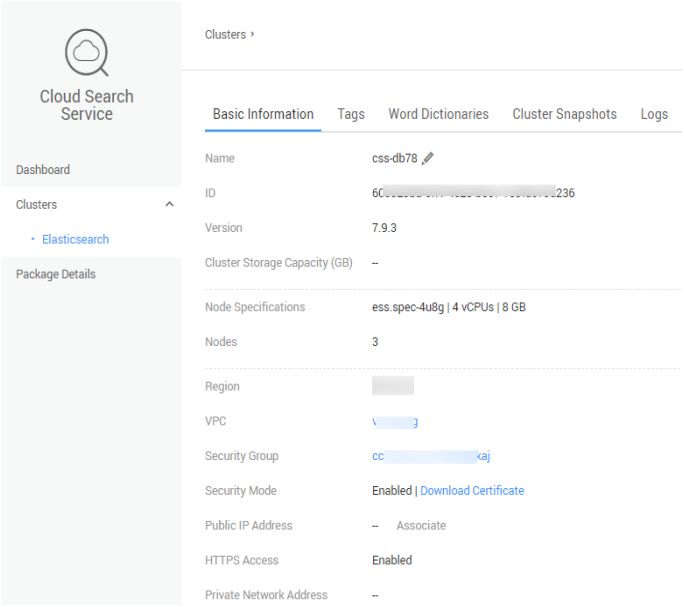
Preparations

The Elasticsearch 6.5.4 and later versions provided by CSS provides the security settings. Once the function is enabled, CSS provides identity authentication, authorization, and encryption for users. Before connecting DLI to the CSS security cluster, you need to perform certain preparations.

1. Select CSS Elasticsearch 6.5.4 or a later cluster version, create a CSS security cluster, and download the security cluster certificate (**CloudSearchService.cer**).

a. Log in to the CSS management console, click **Clusters**, and select the cluster for which you want to create a datasource connection, as shown in **Figure 4-35**.

Figure 4-35 CSS cluster management



b. Click **Download Certificate** next to **Security Mode** to download the security certificate.

2. Use keytool to generate the **keystore** and **truststore** files.

a. Security certificate **CloudSearchService.cer** of the security cluster is required when you use keytool to generate the keystore and truststore files. You can set other keytool parameters as required.

i. Open the cmd command window and run the following command to generate a **keystore** file that contains a private key:

```
keytool -genkeypair -alias certificatekey -keyalg RSA -keystore transport-keystore.jks
```

ii. After the **keystore** and **truststore** files are generated using keytool, you can view the **transport-keystore.jks** file in the folder. Run the following command to verify the **keystore** file and certificate information:

```
keytool -list -v -keystore transport-keystore.jks
```

After you enter the correct keystore password, the corresponding information is displayed.

iii. Run the following commands to create the **truststore.jks** file and verify it:

```
keytool -import -alias certificatekey -file CloudSearchService.cer  -keystore truststore.jks
keytool -list -v -keystore truststore.jks
```

b. Upload the generated **keystore** and **truststore** files to an OBS bucket.

CSS Security Cluster Parameter Configuration

For details about the parameters, see **Table 4-12**. This part describes the precautions for configuring the connection parameters of the CSS security cluster.

```
.option("es.net.http.auth.user", "admin") .option("es.net.http.auth.pass", "***")
```


The parameters are the identity authentication account and password, which are also the account and password for logging in to Kibana.

```
.option("es.net.ssl", "true")
```

- If HTTPS access is enabled for the CSS security cluster, set this parameter to **true** and then set parameters such as the security certificate and file address.
- If HTTPS access is not enabled for the CSS security cluster, set this parameter to **false**. In this case, you do not need to set parameters such as the security certificate and file address.

```
.option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")
.option("es.net.ssl.keystore.pass", "***")
```

Set the location of the **keystore.jks** file and the key for accessing the file. Place the **keystore.jks** file generated in **Preparations** in the OBS bucket, and then enter the AK, SK, and location of the **keystore.jks** file. Enter the key for accessing the file in **es.net.ssl.keystore.pass**.

```
.option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
.option("es.net.ssl.truststore.pass", "***")
```

The parameters in the **truststore.jks** file are basically the same as those in the **keystore.jks** file. You can refer to the preceding procedure to set parameters.

4.3.2.2 Scala Example Code

Prerequisites

A datasource connection has been created on the DLI management console. For details, see **Data Lake Insight User Guide**.

CSS Non-Security Cluster

- Development description
  - Constructing dependency information and creating a Spark session
    i. Import dependencies.

    Maven dependency

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

    Import dependency packages.

    ```
    import org.apache.spark.sql.{Row, SaveMode, SparkSession}
    import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}
    ```

    ii. Create a session.

    ```
    val sparkSession = SparkSession.builder().getOrCreate()
    ```

  - Connecting to data sources through SQL APIs
    i. Create a table to connect to a CSS data source.

    ```
    sparkSession.sql("create table css_table(id int, name string) using css options(
       'es.nodes' 'to-css-1174404221-Y2bKVIqY.datasource.com:9200',
       'es.nodes.wan.only'='true',
       'resource' '/mytest/css')")
    ```


Table 4-12 Parameters for creating a table

Parameter	Description
es.nodes	CSS connection address. You need to create a datasource connection first. For details, see **Enhanced Datasource Connections**. If you have created a basic datasource connection, you can use the returned IP address. If you have created an enhanced datasource connection, use the intranet IP address provided by CSS. The address format is ***IP1*:*PORT1*,*IP2*:*PORT2***.
resource	Name of the resource for the CSS datasource connection name. You can use **/index/type** to specify the resource location (for easier understanding, the **index** may be seen as **database** and **type** as **table**). **NOTE** • In Elasticsearch 6.X, a single index supports only one type, and the type name can be customized. • In Elasticsearch 7.X, a single index uses **_doc** as the type name and cannot be customized. To access Elasticsearch 7.X, set this parameter to **index**.
pushdown	Whether to enable the pushdown function of CSS. The default value is **true**. For tables with a large number of I/O requests, the pushdown function help reduce I/O pressure when the **where** condition is specified.
strict	Whether the CSS pushdown is strict. The default value is **false**. The exact match function can reduce more I/O requests than pushdown.
batch.size.entries	Maximum number of entries that can be inserted in a batch. The default value is **1000**. If the size of a single data record is so large that the number of data records in the bulk storage reaches the upper limit of the data amount in a single batch, the system stops storing data and submits the data based on the **batch.size.bytes** parameter.
batch.size.bytes	Maximum amount of data in a single batch. The default value is **1 MB**. If the size of a single data record is so small that the number of data records in the bulk storage reaches the upper limit of the data amount of a single batch, the system stops storing data and submits the data based on the **batch.size.entries** parameter.

Parameter	Description
es.nodes.wan.only	Whether to access the Elasticsearch node using only the domain name. The default value is **false**. If a basic datasource connection address is used as the **es.nodes**, set this parameter to **true**. If the original internal IP address provided by CSS is used as the **es.nodes**, you do not need to set this parameter or set it to **false**.
es.mapping.id	Document field name that contains the document ID in the Elasticsearch node. **NOTE** • The document ID in the same **/index/type** is unique. If a field that contains a document ID has duplicate values, the document with the duplicate ID will be overwritten when the ES is inserted. • This feature can be used as a fault tolerance solution. When data is being inserted, the DLI job fails and some data has been inserted into Elasticsearch. The data is redundant. If the document ID is set, the previous data will be overwritten when the DLI job is executed again.

NOTE

batch.size.entries and **batch.size.bytes** limit the number of data records and data volume respectively.

ii. Insert data.

```
sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")
```

iii. Query data.

```
val dataFrame = sparkSession.sql("select * from css_table")
dataFrame.show()
```

Before data is inserted:

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
+---+----+\n
```

Response:

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
| 13|John|\n
| 22| Bob|\n
+---+----+\n
```

iv. Delete the datasource connection table.

```
sparkSession.sql("drop table css_table")
```


- Connecting to data sources through DataFrame APIs

  i. Set connection parameters.

  ```
  val resource = "/mytest/css"
  val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"
  ```

  ii. Create a schema and add data to it.

  ```
  val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name", StringType, false)))
  val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))
  ```

  iii. Import data to CSS.

  ```
  val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)
  dataFrame_1.write
    .format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .mode(SaveMode.Append)
    .save()
  ```

  NOTE

  The value of **SaveMode** can be one of the following:

  - **ErrorIfExis**: If the data already exists, the system throws an exception.
  - **Overwrite**: If the data already exists, the original data will be overwritten.
  - **Append**: If the data already exists, the system saves the new data.
  - **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

  iv. Read data from CSS.

  ```
  val dataFrameR = sparkSession.read.format("css").option("resource",resource).option("es.nodes", nodes).load()
  dataFrameR.show()
  ```

  Before data is inserted:

  ```
  +---+----+\n
  | id|name|\n
  +---+----+\n
  |  1|John|\n
  | 22| Bob|\n
  +---+----+\n
  ```

  Response:

  ```
  +---+----+\n
  | id|name|\n
  +---+----+\n
  |  1|John|\n
  | 12|John|\n
  | 21| Bob|\n
  | 22| Bob|\n
  +---+----+\n
  ```

- Submitting a Spark job

  i. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

  ii. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

NOTE

- For Spark 2.3.2 (soon to be take offline) or 2.4.5, set **Module** to **sys.datasource.css** when submitting a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

- Complete example code
  - Maven dependency

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  - Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.SparkSession

object Test_SQL_CSS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a DLI data table for DLI-associated CSS
    sparkSession.sql("create table css_table(id long, name string) using css options(
    'es.nodes' = 'to-css-1174404217-QG2SwbVV.datasource.com:9200',
    'es.nodes.wan.only' = 'true',
    'resource' = '/mytest/css')")

    //*****************************SQL model***********************************
    // Insert data into the DLI data table
    sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")

    // Read data from DLI data table
    val dataFrame = sparkSession.sql("select * from css_table")
    dataFrame.show()

    // drop table
    sparkSession.sql("drop table css_table")

    sparkSession.close()
  }
}
```

  - Connecting to data sources through DataFrame APIs

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession};
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType};

object Test_SQL_CSS {
  def main(args: Array[String]): Unit = {
    //Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    //*****************************DataFrame model***********************************
    // Setting the /index/type of CSS
    val resource = "/mytest/css"
```



```
    // Define the cross-origin connection address of the CSS cluster
    val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"

    //Setting schema
    val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name",
StringType, false)))

    // Construction data
    val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))

    // Create a DataFrame from RDD and schema
    val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)

   //Write data to the CSS
   dataFrame_1.write.format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .mode(SaveMode.Append)
    .save()

    //Read data
    val dataFrameR = sparkSession.read.format("css").option("resource",
resource).option("es.nodes", nodes).load()
    dataFrameR.show()

    spardSession.close()
  }
}
```

CSS Security Cluster

- Development description
  - Constructing dependency information and creating a Spark session
    i. Import dependencies.

       Maven dependency

       ```
       <dependency>
         <groupId>org.apache.spark</groupId>
         <artifactId>spark-sql_2.11</artifactId>
         <version>2.3.2</version>
       </dependency>
       ```

       Import dependency packages.

       ```
       import org.apache.spark.sql.{Row, SaveMode, SparkSession}
       import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}
       ```

    ii. Create a session and set the AKs and SKs.

       > NOTE
       >
       > Hard-coded or plaintext AK and SK pose significant security risks. To ensure security, encrypt your AK and SK, store them in configuration files or environment variables, and decrypt them when needed.

       ```
       val sparkSession = SparkSession.builder().getOrCreate()
       sparkSession.conf.set("fs.obs.access.key", ak)
       sparkSession.conf.set("fs.obs.secret.key", sk)
       sparkSession.conf.set("fs.obs.endpoint", enpoint)
       sparkSession.conf.set("fs.obs.connecton.ssl.enabled", "false")
       ```

  - Connecting to data sources through SQL APIs
    i. Create a table to connect to a CSS data source.

       ```
       sparkSession.sql("create table css_table(id int, name string) using css options(
          'es.nodes' 'to-css-1174404221-Y2bKVIqY.datasource.com:9200',
          'es.nodes.wan.only'='true',
          'resource'='/mytest/css',
           'es.net.ssl'='true',
          'es.net.ssl.keystore.location'='obs://Bucket name/path/transport-keystore.jks',
       ```



```
'es.net.ssl.keystore.pass'='***',
'es.net.ssl.truststore.location'='obs://Bucket name/path/truststore.jks',
'es.net.ssl.truststore.pass'='***',
'es.net.http.auth.user'='admin',
'es.net.http.auth.pass'='***')")
```

Table 4-13 Parameters for creating a table

Parameter	Description
es.nodes	CSS connection address. You need to create a datasource connection first. For details, see **Enhanced Datasource Connections**. If you have created a basic datasource connection, you can use the returned IP address. If you have created an enhanced datasource connection, use the intranet IP address provided by CSS. The address format is ***IP1*:*PORT1*,*IP2*:*PORT2***.
resource	Name of the resource for the CSS datasource connection name. You can use **/index/type** to specify the resource location (for easier understanding, the **index** may be seen as **database** and **type** as **table**). **NOTE** 1. In Elasticsearch 6.*X*, a single index supports only one type, and the type name can be customized. 2. In Elasticsearch 7.*X*, a single index uses **_doc** as the type name and cannot be customized. To access Elasticsearch 7.*X*, set this parameter to **index**.
pushdown	Whether to enable the pushdown function of CSS. The default value is **true**. For tables with a large number of I/O requests, the pushdown function help reduce I/O pressure when the **where** condition is specified.
strict	Whether the CSS pushdown is strict. The default value is **false**. The exact match function can reduce more I/O requests than pushdown.
batch.size.entries	Maximum number of entries that can be inserted in a batch. The default value is **1000**. If the size of a single data record is so large that the number of data records in the bulk storage reaches the upper limit of the data amount in a single batch, the system stops storing data and submits the data based on the **batch.size.bytes** parameter.

Parameter	Description
batch.size.bytes	Maximum amount of data in a single batch. The default value is **1 MB**. If the size of a single data record is so small that the number of data records in the bulk storage reaches the upper limit of the data amount of a single batch, the system stops storing data and submits the data based on the **batch.size.entries** parameter.
es.nodes.wan.only	Whether to access the Elasticsearch node using only the domain name. The default value is **false**. If a basic datasource connection address is used as the **es.nodes**, set this parameter to **true**. If the original internal IP address provided by CSS is used as the **es.nodes**, you do not need to set this parameter or set it to **false**.
es.mapping.id	Document field name that contains the document ID in the Elasticsearch node. **NOTE** • The document ID in the same **/index/type** is unique. If a field that contains a document ID has duplicate values, the document with the duplicate ID will be overwritten when the ES is inserted. • This feature can be used as a fault tolerance solution. When data is being inserted, the DLI job fails and some data has been inserted into Elasticsearch. The data is redundant. If the document ID is set, the previous data will be overwritten when the DLI job is executed again.
es.net.ssl	Whether to connect to the security CSS cluster. The default value is **false**.
es.net.ssl.keystore.location	OBS bucket location of the **keystore** file generated by the security CSS cluster certificate.
es.net.ssl.keystore.pass	Password of the **keystore** file generated by the security CSS cluster certificate.
es.net.ssl.truststore.location	OBS bucket location of the **truststore** file generated by the security CSS cluster certificate.
es.net.ssl.truststore.pass	Password of the **truststore** file generated by the security CSS cluster certificate.
es.net.http.auth.user	Username of the security CSS cluster.
es.net.http.auth.pass	Password of the security CSS cluster.

NOTE

batch.size.entries and **batch.size.bytes** limit the number of data records and data volume respectively.

ii. Insert data.

```
sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")
```

iii. Query data.

```
val dataFrame = sparkSession.sql("select * from css_table")
dataFrame.show()
```

Before data is inserted:

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
+---+----+\n
```

Response:

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
| 13|John|\n
| 22| Bob|\n
+---+----+\n
```

iv. Delete the datasource connection table.

```
sparkSession.sql("drop table css_table")
```

– Connecting to data sources through DataFrame APIs

i. Set connection parameters.

```
val resource = "/mytest/css"
val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"
```

ii. Create a schema and add data to it.

```
val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name", StringType, false)))
val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))
```

iii. Import data to CSS.

```
val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)
dataFrame_1.write
  .format("css")
  .option("resource", resource)
  .option("es.nodes", nodes)
  .option("es.net.ssl", "true")
  .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")
  .option("es.net.ssl.keystore.pass", "***")
  .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
  .option("es.net.ssl.truststore.pass", "***")
  .option("es.net.http.auth.user", "admin")
  .option("es.net.http.auth.pass", "***")
  .mode(SaveMode.Append)
  .save()
```


NOTE

The value of **Mode** can be one of the following:

- **ErrorIfExis**: If the data already exists, the system throws an exception.
- **Overwrite**: If the data already exists, the original data will be overwritten.
- **Append**: If the data already exists, the system saves the new data.
- **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

iv. Read data from CSS.

```
val dataFrameR = sparkSession.read.format("css")
      .option("resource",resource)
      .option("es.nodes", nodes)
      .option("es.net.ssl", "true")
      .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")
      .option("es.net.ssl.keystore.pass", "***")
      .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
      .option("es.net.ssl.truststore.pass", "***")
      .option("es.net.http.auth.user", "admin")
      .option("es.net.http.auth.pass", "***")
      .load()
dataFrameR.show()
```

Before data is inserted:

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
| 22| Bob|\n
+---+----+\n
```

Response:

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
| 12|John|\n
| 21| Bob|\n
| 22| Bob|\n
+---+----+\n
```

- Submitting a Spark job

  i. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

  ii. In the Spark job editor, select the corresponding dependency module and execute the Spark job. For details about console operations, see **Creating a Spark Job**. For API operations, see **Creating a Batch Processing Job**.

NOTE

- When submitting a job, you need to specify a dependency module named **sys.datasource.css**.
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the **modules** parameter in **Request parameters** of **Creating a Batch Processing Job** in the *Data Lake Insight API Reference*.

- Complete example code
  - Maven dependency

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  - Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.SparkSession

object csshttpstest {
  def main(args: Array[String]): Unit = {
    //Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    // Create a DLI data table for DLI-associated CSS
    sparkSession.sql("create table css_table(id long, name string) using css options('es.nodes' =
'192.168.6.204:9200','es.nodes.wan.only' = 'false','resource' = '/
mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'obs://xietest1/lzq/
keystore.jks','es.net.ssl.keystore.pass' = '**','es.net.ssl.truststore.location'='obs://xietest1/lzq/
truststore.jks','es.net.ssl.truststore.pass'='**','es.net.http.auth.user'='admin','es.net.http.auth.pass'='*
*')")

    //*****************************SQL model***********************************
    // Insert data into the DLI data table
    sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")

    // Read data from DLI data table
    val dataFrame = sparkSession.sql("select * from css_table")
    dataFrame.show()

    // drop table
    sparkSession.sql("drop table css_table")

    sparkSession.close()
  }
}
```

  - Connecting to data sources through DataFrame APIs

NOTE

Hard-coded or plaintext AK and SK pose significant security risks. To ensure security, encrypt your AK and SK, store them in configuration files or environment variables, and decrypt them when needed.

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession};
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType};

object Test_SQL_CSS {
  def main(args: Array[String]): Unit = {
    //Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    sparkSession.conf.set("fs.obs.access.key", ak)
    sparkSession.conf.set("fs.obs.secret.key", sk)

    //*****************************DataFrame model***********************************
    // Setting the /index/type of CSS
```



```
    val resource = "/mytest/css"

    // Define the cross-origin connection address of the CSS cluster
    val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"

    //Setting schema
    val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name",
StringType, false)))

    // Construction data
    val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))

    // Create a DataFrame from RDD and schema
    val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)

   //Write data to the CSS
   dataFrame_1.write .format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .option("es.net.ssl", "true")
    .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")
    .option("es.net.ssl.keystore.pass", "***")
    .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")
    .option("es.net.http.auth.user", "admin")
    .option("es.net.http.auth.pass", "***")
    .mode(SaveMode.Append)
    .save();

    //Read data
    val dataFrameR = sparkSession.read.format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .option("es.net.ssl", "true")
    .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")
    .option("es.net.ssl.keystore.pass", "***")
    .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")
    .option("es.net.http.auth.user", "admin")
    .option("es.net.http.auth.pass", "***")
    .load()
    dataFrameR.show()

    spardSession.close()
  }
}
```

4.3.2.3 PySpark Example Code

Prerequisites

A datasource connection has been created on the DLI management console. For details, see **Data Lake Insight User Guide**.

CSS Non-Security Cluster

- Development description
  - Code implementation
    i. Import dependency packages.

```
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, Row
from pyspark.sql import SparkSession
```

    ii. Create a session.

```
sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
```


- Connecting to data sources through DataFrame APIs

  i. Set connection parameters.

  ```
  resource = "/mytest"
  nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"
  ```

  NOTE

  resource indicates the name of the resource associated with the CSS. You can specify the resource location in */index/type* format. (The **index** can be the database and **type** the table.)

  - In Elasticsearch 6.X, a single index supports only one type, and the type name can be customized.
  - In Elasticsearch 7.X, a single index uses **_doc** as the type name and cannot be customized. To access Elasticsearch 7.X, set this parameter to **index**.

  ii. Create a schema and add data to it.

  ```
  schema = StructType([StructField("id", IntegerType(), False),
                       StructField("name", StringType(), False)])
  rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])
  ```

  iii. Construct a DataFrame.

  ```
  dataFrame = sparkSession.createDataFrame(rdd, schema)
  ```

  iv. Save data to CSS.

  ```
  dataFrame.write.format("css").option("resource", resource).option("es.nodes",
  nodes).mode("Overwrite").save()
  ```

  NOTE

  The options of **mode** can be one of the following:

  - **ErrorIfExis**: If the data already exists, the system throws an exception.
  - **Overwrite**: If the data already exists, the original data will be overwritten.
  - **Append**: If the data already exists, the system saves the new data.
  - **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

  v. Read data from CSS.

  ```
  jdbcDF = sparkSession.read.format("css").option("resource", resource).option("es.nodes",
  nodes).load()
  jdbcDF.show()
  ```

  vi. View the operation result.

  ```
  +---+----+
  | id|name|
  +---+----+
  |  2| Bob|
  |  1|John|
  +---+----+
  ```

- Connecting to data sources through SQL APIs

  i. Create a table to connect to a CSS data source.

  ```
  sparkSession.sql(
     "create table css_table(id long, name string) using css options(
     'es.nodes'='to-css-1174404953-hDTx3UPK.datasource.com:9200',
     'es.nodes.wan.only'='true',
     'resource'='/mytest')")
  ```


NOTE

For details about the parameters for creating a CSS datasource connection table, see **Table 4-12**.

ii. Insert data.

```
sparkSession.sql("insert into css_table values(3,'tom')")
```

iii. Query data.

```
jdbcDF = sparkSession.sql("select * from css_table")
jdbcDF.show()
```

iv. View the operation result.

```
+---+----+
| id|name|
+---+----+
|  3| tom|
|  2| Bob|
|  1|John|
+---+----+
```

– Submitting a Spark job

i. Upload the Python code file to the OBS bucket.

ii. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

NOTE

- For Spark 2.3.2 (soon to be take offline) or 2.4.5, set **Module** to **sys.datasource.css** when submitting a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

- Complete example code

  – Connecting to data sources through DataFrame APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import Row, StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()

  # Setting cross-source connection parameters
  resource = "/mytest"
  nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),
                       StructField("name", StringType(), False)])
```



```
# Construction data
rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])

# Create a DataFrame from RDD and schema
dataFrame = sparkSession.createDataFrame(rdd, schema)

# Write data to the CSS
dataFrame.write.format("css").option("resource", resource).option("es.nodes",
nodes).mode("Overwrite").save()

# Read data
jdbcDF = sparkSession.read.format("css").option("resource", resource).option("es.nodes",
nodes).load()
jdbcDF.show()

# close session
sparkSession.stop()
```

- Connecting to data sources through SQL APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()

  # Create a DLI data table for DLI-associated CSS
  sparkSession.sql(
      "create table css_table(id long, name string) using css options( \
      'es.nodes'='to-css-1174404953-hDTx3UPK.datasource.com:9200',\
      'es.nodes.wan.only'='true',\
      'resource'='/mytest')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into css_table values(3,'tom')")

  # Read data from DLI data table
  jdbcDF = sparkSession.sql("select * from css_table")
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

CSS Security Cluster

- Development description
  - Code implementation
    i. Import dependency packages.

```
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, Row
from pyspark.sql import SparkSession
```

    ii. Create a session and set the AKs and SKs.

> NOTE
>
> Hard-coded or plaintext AK and SK pose significant security risks. To ensure security, encrypt your AK and SK, store them in configuration files or environment variables, and decrypt them when needed.

```
sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
sparkSession.conf.set("fs.obs.access.key", ak)
sparkSession.conf.set("fs.obs.secret.key", sk)
sparkSession.conf.set("fs.obs.endpoint", enpoint)
sparkSession.conf.set("fs.obs.connecton.ssl.enabled", "false")
```

  - Connecting to data sources through DataFrame APIs

i. Set connection parameters.

```
resource = "/mytest";
nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"
```

NOTE

resource indicates the name of the resource associated with the CSS. You can specify the resource location in */index/type* format. (The **index** can be the database and **type** the table.)

- In Elasticsearch 6.X, a single index supports only one type, and the type name can be customized.
- In Elasticsearch 7.X, a single index uses **_doc** as the type name and cannot be customized. To access Elasticsearch 7.X, set this parameter to **index**.

ii. Create a schema and add data to it.

```
schema = StructType([StructField("id", IntegerType(), False),
                     StructField("name", StringType(), False)])
rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])
```

iii. Construct a DataFrame.

```
dataFrame = sparkSession.createDataFrame(rdd, schema)
```

iv. Save data to CSS.

```
dataFrame.write.format("css")
  .option("resource", resource)
  .option("es.nodes", nodes)
  .option("es.net.ssl", "true")
  .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")
  .option("es.net.ssl.keystore.pass", "***")
  .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
  .option("es.net.ssl.truststore.pass", "***")
  .option("es.net.http.auth.user", "admin")
  .option("es.net.http.auth.pass", "***")
  .mode("Overwrite")
  .save()
```

NOTE

The options of **mode** can be one of the following:

- **ErrorIfExis**: If the data already exists, the system throws an exception.
- **Overwrite**: If the data already exists, the original data will be overwritten.
- **Append**: If the data already exists, the system saves the new data.
- **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

v. Read data from CSS.

```
jdbcDF = sparkSession.read.format("css")\
  .option("resource", resource)\
  .option("es.nodes", nodes)\
  .option("es.net.ssl", "true")\
  .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")\
  .option("es.net.ssl.keystore.pass", "***")\
  .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")\
  .option("es.net.ssl.truststore.pass", "***")\
  .option("es.net.http.auth.user", "admin")\
  .option("es.net.http.auth.pass", "***")\
  .load()
jdbcDF.show()
```

vi. View the operation result.


```
+---+----+
| id|name|
+---+----+
|  2| Bob|
|  1|John|
+---+----+
```

- Connecting to data sources through SQL APIs
  i. Create a table to connect to a CSS data source.

```
sparkSession.sql(
      "create table css_table(id long, name string) using css options(\
      'es.nodes'='to-css-1174404953-hDTx3UPK.datasource.com:9200',\
      'es.nodes.wan.only'='true',\
      'resource'='/mytest',\
   'es.net.ssl'='true',\
   'es.net.ssl.keystore.location'='obs://Bucket name/path/transport-keystore.jks',\
   'es.net.ssl.keystore.pass'='***',\
   'es.net.ssl.truststore.location'='obs://Bucket name/path/truststore.jks',\
   'es.net.ssl.truststore.pass'='***',\
   'es.net.http.auth.user'='admin',\
   'es.net.http.auth.pass'='***')")
```

  NOTE

  For details about the parameters for creating a CSS datasource connection table, see **Table 4-12**.

  ii. Insert data.

```
sparkSession.sql("insert into css_table values(3,'tom')")
```

  iii. Query data.

```
jdbcDF = sparkSession.sql("select * from css_table")
jdbcDF.show()
```

  iv. View the operation result.

```
+---+----+
| id|name|
+---+----+
|  3| tom|
|  2| Bob|
|  1|John|
+---+----+
```

- Submitting a Spark job
  i. Upload the Python code file to DLI. For details about console operations, see **Creating a Package**. For API operations, see **Uploading a Package Group**.
  ii. In the Spark job editor, select the corresponding dependency module and execute the Spark job. For details about console operations, see **Creating a Spark Job**. For API operations, see **Creating a Batch Processing Job**.

  NOTE

  - When submitting a job, you need to specify a dependency module named **sys.datasource.css**.
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the **modules** parameter in **Request parameters** of **Creating a Batch Processing Job** in the *Data Lake Insight API Reference*.

- Complete example code
  - Connecting to data sources through DataFrame APIs

    NOTE

    Hard-coded or plaintext AK and SK pose significant security risks. To ensure security, encrypt your AK and SK, store them in configuration files or environment variables, and decrypt them when needed.

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import Row, StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
  sparkSession.conf.set("fs.obs.access.key", ak)
  sparkSession.conf.set("fs.obs.secret.key", sk)
  sparkSession.conf.set("fs.obs.endpoint", enpoint)
  sparkSession.conf.set("fs.obs.connecton.ssl.enabled", "false")

  # Setting cross-source connection parameters
  resource = "/mytest";
  nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),
                       StructField("name", StringType(), False)])

  # Construction data
  rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(rdd, schema)

  # Write data to the CSS
  dataFrame.write.format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .option("es.net.ssl", "true")
    .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")
    .option("es.net.ssl.keystore.pass", "***")
    .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")
    .option("es.net.http.auth.user", "admin")
    .option("es.net.http.auth.pass", "***")
    .mode("Overwrite")
    .save()

  # Read data
  jdbcDF = sparkSession.read.format("css")\
    .option("resource", resource)\
    .option("es.nodes", nodes)\
    .option("es.net.ssl", "true")\
    .option("es.net.ssl.keystore.location", "obs://Bucket name/path/transport-keystore.jks")\
    .option("es.net.ssl.keystore.pass", "***")\
    .option("es.net.ssl.truststore.location", "obs://Bucket name/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")\
    .option("es.net.http.auth.user", "admin")\
    .option("es.net.http.auth.pass", "***")\
    .load()
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

  - Connecting to data sources through SQL APIs


```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession
import os

if __name__ == "__main__":

  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
  # Create a DLI data table for DLI-associated CSS
  sparkSession.sql("create table css_table(id int, name string) using css options(\
                    'es.nodes'='192.168.6.204:9200',\
                    'es.nodes.wan.only'='true',\
                    'resource'='/mytest',\
                    'es.net.ssl'='true',\
                    'es.net.ssl.keystore.location' = 'obs://xietest1/lzq/keystore.jks',\
                    'es.net.ssl.keystore.pass' = '**',\
                    'es.net.ssl.truststore.location'='obs://xietest1/lzq/truststore.jks',\
                    'es.net.ssl.truststore.pass'='**',\
                    'es.net.http.auth.user'='admin',\
                    'es.net.http.auth.pass'='**')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into css_table values(3,'tom')")

  # Read data from DLI data table
  jdbcDF = sparkSession.sql("select * from css_table")
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

4.3.2.4 Java Example Code

Prerequisites

A datasource connection has been created on the DLI management console. For details, see **Enhanced Datasource Connections**.

CSS Non-Security Cluster

- Development description
  - Code implementation
    - Constructing dependency information and creating a Spark session
      1) Import dependencies.

         Maven dependency

         ```
         <dependency>
                 <groupId>org.apache.spark</groupId>
                 <artifactId>spark-sql_2.11</artifactId>
                 <version>2.3.2</version>
         </dependency>
         ```

         Import dependency packages.

         ```
         import org.apache.spark.sql.SparkSession;
         ```

      2) Create a session.

         ```
         SparkSession sparkSession = SparkSession.builder().appName("datasource-
         css").getOrCreate();
         ```

  - Connecting to data sources through SQL APIs
    i. Create a table to connect to a CSS data source.


```
sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes'
= '192.168.9.213:9200', 'es.nodes.wan.only' = 'true','resource' ='/mytest')");
```

ii. Insert data.

```
sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");
```

iii. Query data.

```
sparkSession.sql("select * from css_table").show();
```

iv. Delete the datasource connection table.

```
sparkSession.sql("drop table css_table");
```

– Submitting a Spark job

i. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

ii. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

NOTE

- For Spark 2.3.2 (soon to be take offline) or 2.4.5, set **Module** to **sys.datasource.css** when submitting a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

- Complete example code

  – Maven dependency

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.3.2</version>
</dependency>
```

  – Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.*;

public class java_css_unsecurity {

    public static void main(String[] args) {
        SparkSession sparkSession = SparkSession.builder().appName("datasource-css-
unsecurity").getOrCreate();

        // Create a DLI data table for DLI-associated CSS
        sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes'
= '192.168.15.34:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/mytest')");

        //*****************************SQL model***********************************
        // Insert data into the DLI data table
        sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");

        // Read data from DLI data table
        sparkSession.sql("select * from css_table").show();

        // drop table
        sparkSession.sql("drop table css_table");
```



```
        sparkSession.close();
    }
}
```

CSS Security Cluster

- Preparations

  Generate the **keystore.jks** and **truststore.jks** files and upload them to the OBS bucket. For details, see **CSS Security Cluster Configuration**.

- Description of the development with HTTPS disabled

  If HTTPS is disabled, **keystore.jks** and **truststore.jks** files are not required. You only need to set SSL access parameters and credentials.

  – Constructing dependency information and creating a Spark session

    i. Import dependencies.

    Maven dependency

    ```
    <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
    </dependency>
    ```

    Import dependency packages.

    ```
    import org.apache.spark.sql.SparkSession;
    ```

    ii. Create a session.

    ```
    SparkSession sparkSession = SparkSession.builder().appName("datasource-
    css").getOrCreate();
    ```

  – Connecting to data sources through SQL APIs

    i. Create a table to connect to a CSS data source.

    ```
    sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes'
    = '192.168.9.213:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/
    mytest','es.net.ssl'='false','es.net.http.auth.user'='admin','es.net.http.auth.pass'='*******')");
    ```

    NOTE

    - For details about the parameters for creating a CSS datasource connection table, see **Table 4-12**.
    - In the preceding example, HTTPS access is disabled for the CSS security cluster. Therefore, you need to set **es.net.ssl** to **false**. **es.net.http.auth.user** and **es.net.http.auth.pass** are the username and password set during cluster creation, respectively.

    ii. Insert data.

    ```
    sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");
    ```

    iii. Query data.

    ```
    sparkSession.sql("select * from css_table").show();
    ```

    iv. Delete the datasource connection table.

    ```
    sparkSession.sql("drop table css_table");
    ```

  – Submitting a Spark job

    i. Generate a JAR package based on the code file and upload the package to DLI.

    For details about console operations, see **Creating a Package**. For details about API operations, see **Uploading a Package Group**.

    ii. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

For details about console operations, see **Creating a Spark Job**. For details about API operations, see **Creating a Batch Processing Job**.

NOTE

- When submitting a job, you need to specify a dependency module named **sys.datasource.css**.
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the **modules** parameter in **Request parameters** of **Creating a Batch Processing Job** in the *Data Lake Insight API Reference*.

– Complete example code

▪ Maven dependency

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.3.2</version>
</dependency>
```

- Description of development with HTTPS enabled
  – Constructing dependency information and creating a Spark session

    i. Import dependencies.

    Maven dependency

    ```
    <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
    </dependency>
    ```

    Import dependency packages.

    ```
    import org.apache.spark.SparkFiles;
    import org.apache.spark.sql.SparkSession;
    import java.io.IOException;
    import java.nio.file.Files;
    import java.nio.file.Paths;
    ```

    ii. Create a session.

    ```
    SparkSession sparkSession = SparkSession.builder().appName("datasource-
    css").getOrCreate();
    ```

    iii. Copy the certificate.

    ```
        sparkSession.sparkContext().addFile("obs://Bucket name/Address/transport-
    keystore.jks");
        sparkSession.sparkContext().addFile("obs://Bucket name/Address/truststore.jks");

          // Obtain the path of the current working directory.
          String pathUser = System.getProperty("user.dir");
          System.out.println("path_user is " + pathUser);

          // Obtain the file name.
          String esTransportKeystoreFileName = SparkFiles.get("transport-keystore.jks");
          String esTruststoreFileName = SparkFiles.get("truststore.jks");

          System.out.println("esTransportKeystoreFileName is " +
    esTransportKeystoreFileName);
          System.out.println("esTruststoreFileName is " + esTruststoreFileName);
          // Combine the file path.
          String esTransportKeystoreLocalPath = pathUser + "/" + "transport-keystore.jks";
          String esTruststoreLocalPath = pathUser + "/" + "truststore.jks";

          System.out.println("esTransportKeystoreLocalPath is " +
    esTransportKeystoreLocalPath);
    ```



```
        System.out.println("esTruststoreLocalPath is " + esTruststoreLocalPath);
        try {
            // Copy the keystore file.
            copyFile(esTransportKeystoreFileName, esTransportKeystoreLocalPath);
            // Copy the truststore file.
            copyFile(esTruststoreFileName, esTruststoreLocalPath);
            // Wait for a few minutes.
            Thread.sleep(2000);

            System.out.println("Files copied successfully:");
            System.out.println("es_transport-keystore.jks: " + esTransportKeystoreLocalPath);
            System.out.println("es_truststore.jks: " + esTruststoreLocalPath);
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
```

- Connecting to data sources through SQL APIs

  i. Create a table to connect to a CSS data source.

  ```
  sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes' = '192.168.13.189:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'file://" + esTransportKeystoreLocalPath + "','es.net.ssl.keystore.pass' = '**', 'es.net.ssl.truststore.location'='file://" + esTruststoreLocalPath + "', 'es.net.ssl.truststore.pass'='***','es.net.http.auth.user'='admin','es.net.http.auth.pass'='**')");
  ```

  NOTE

  For details about the parameters for creating a CSS datasource connection table, see **Table 4-12**.

  ii. Insert data.

  ```
  sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");
  ```

  iii. Query data.

  ```
  sparkSession.sql("select * from css_table").show();
  ```

  iv. Delete the datasource connection table.

  ```
  sparkSession.sql("drop table css_table");
  ```

- Submitting a Spark job

  i. Generate a JAR package based on the code file and upload the package to DLI.

  For details about console operations, see **Creating a Package**. For details about API operations, see **Uploading a Package Group**.

  ii. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  For details about console operations, see **Creating a Spark Job**. For details about API operations, see **Creating a Batch Processing Job**.

  NOTE

  - When submitting a job, you need to specify a dependency module named **sys.datasource.css**.
  - For details about how to submit a job on the console, see **Parameters for selecting dependency resources** in the **Data Lake Insight User Guide**.
  - For details about how to submit a job through an API, see the **modules** parameter in **Request parameters** of **Creating a Batch Processing Job** in the *Data Lake Insight API Reference*.

- Complete example code

- Maven dependency

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.3.2</version>
</dependency>
```

- Connecting to data sources through SQL APIs

```
import org.apache.spark.SparkFiles;
import org.apache.spark.sql.SparkSession;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class java_css_security_httpson {
    public static void main(String[] args) {
        SparkSession sparkSession = SparkSession.builder().appName("datasource-
css").getOrCreate();

        sparkSession.sparkContext().addFile("obs://Bucket name/Address/transport-
keystore.jks");
        sparkSession.sparkContext().addFile("obs://Bucket name/Address/css/truststore.jks");

        // Obtain the path of the current working directory.
        String pathUser = System.getProperty("user.dir");
        System.out.println("path_user is " + pathUser);

        // Obtain the file name.
        String esTransportKeystoreFileName = SparkFiles.get("transport-keystore.jks");
        String esTruststoreFileName = SparkFiles.get("truststore.jks");

        System.out.println("esTransportKeystoreFileName is " +
esTransportKeystoreFileName);
        System.out.println("esTruststoreFileName is " + esTruststoreFileName);
        // Combine the file path.
        String esTransportKeystoreLocalPath = pathUser + "/" + "transport-keystore.jks";
        String esTruststoreLocalPath = pathUser + "/" + "truststore.jks";

        System.out.println("esTransportKeystoreLocalPath is " +
esTransportKeystoreLocalPath);
        System.out.println("esTruststoreLocalPath is " + esTruststoreLocalPath);
        try {
            // Copy the keystore file.
            copyFile(esTransportKeystoreFileName, esTransportKeystoreLocalPath);
            // Copy the truststore file.
            copyFile(esTruststoreFileName, esTruststoreLocalPath);
            // Wait for a few minutes.
            Thread.sleep(2000);

            System.out.println("Files copied successfully:");
            System.out.println("es_transport-keystore.jks: " + esTransportKeystoreLocalPath);
            System.out.println("es_truststore.jks: " + esTruststoreLocalPath);
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }

        // Create a DLI data table for DLI-associated CSS
        sparkSession.sql("create table css_table(id long, name string) using css
options( 'es.nodes' = '192.168.13.189:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/
mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'file://" +
esTransportKeystoreLocalPath + "','es.net.ssl.keystore.pass' =
'**','es.net.ssl.truststore.location'='file://" + esTruststoreLocalPath +
"','es.net.ssl.truststore.pass'='**','es.net.http.auth.user'='admin','es.net.http.auth.pass'='**')");

        //*****************************SQL model*********************************
        // Insert data into the DLI data table
        sparkSession.sql("insert into css_table values(34, 'Yuan'),(28, 'Kids')");

        // Read data from DLI data table
```



```
        sparkSession.sql("select * from css_table").show();

        // drop table
        sparkSession.sql("drop table css_table");

        sparkSession.close();
    }
    private static void copyFile(String sourcePath, String destinationPath) throws
IOException {
        // Copy a file from remote storage to local storage.
        byte[] fileContent = Files.readAllBytes(Paths.get(sourcePath));
        Files.write(Paths.get(destinationPath), fileContent);
    }
}
```

4.3.3 Connecting to GaussDB(DWS)

4.3.3.1 Scala Example Code

Scenario

This section provides Scala example code that demonstrates how to use a Spark job to access data from the GaussDB(DWS) data source.

A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

NOTE

Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

Preparations

Constructing dependency information and creating a Spark session

1. Import dependencies

   Involved Maven dependency

   ```
   <dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-sql_2.11</artifactId>
     <version>2.3.2</version>
   </dependency>
   ```

   Import dependency packages.

   ```
   import java.util.Properties
   import org.apache.spark.sql.{Row,SparkSession}
   import org.apache.spark.sql.SaveMode
   ```

2. Create a session.

   ```
   val sparkSession = SparkSession.builder().getOrCreate()
   ```

Accessing a Data Source Using a SQL API

1. Create a table to connect to a GaussDB(DWS) data source.

   ```
   sparkSession.sql(
     "CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (
       'url'='jdbc:postgresql://to-dws-1174404209-cA37siB6.datasource.com:8000/postgres',
       'dbtable'='customer',
       'user'='dbadmin',
   ```

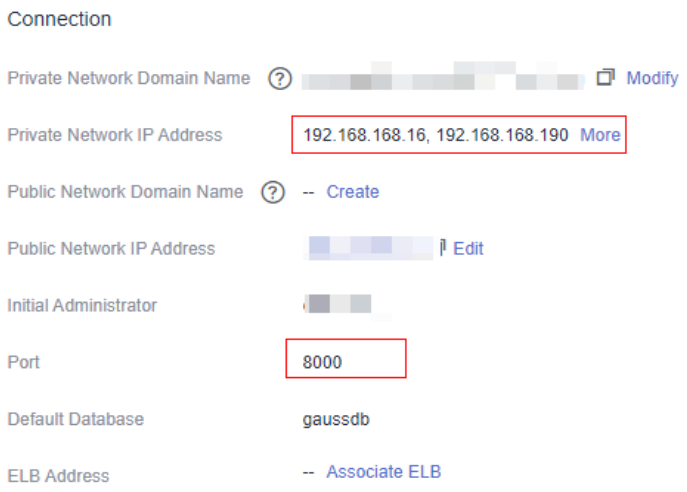
```
    'passwdauth'='######'// Name of the datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for the job.
)"
)
```

Table 4-14 Parameters for creating a table

Parameter	Description
url	To obtain a GaussDB(DWS) IP address, you need to create a datasource connection first. Refer to *Data Lake Insight User Guide* for more information. After a basic datasource connection is created, the returned IP address is used. After an enhanced datasource connection is created, you can use the JDBC connection string (intranet) provided by GaussDB(DWS) or the intranet IP address and port number to connect to GaussDB(DWS). The format is ***protocol header*://*internal IP address*:*internal network port number*/*database name***, for example: **jdbc:postgresql://192.168.0.77:8000/postgres**. For details about how to obtain the value, see *GaussDB(DWS) cluster information*. **NOTE** The GaussDB(DWS) IP address is in the following format: ***protocol header*://*IP address*:*port number*/*database name*** Example: jdbc:postgresql://to-dws-1174405119-ihlUr78j.datasource.com:8000/postgres If you want to connect to a database created in GaussDB(DWS), change **postgres** to the corresponding database name in this connection.
passwdauth	Name of datasource authentication of the password type created on DLI. If datasource authentication is used, you do not need to set the username and password for jobs.
dbtable	Tables in the PostgreSQL database.
partitionColumn	This parameter is used to set the numeric field used concurrently when data is read. **NOTE** • The **partitionColumn**, **lowerBound**, **upperBound**, and **numPartitions** parameters must be set at the same time. • To improve the concurrent read performance, you are advised to use auto-increment columns.
lowerBound	Minimum value of a column specified by **partitionColumn**. The value is contained in the returned result.
upperBound	Maximum value of a column specified by **partitionColumn**. The value is not contained in the returned result.

Parameter	Description
numPartitions	Number of concurrent read operations. **NOTE** When data is read, **lowerBound** and **upperBound** are evenly allocated to each task to obtain data. Example: 'partitionColumn'='id', 'lowerBound'='0', 'upperBound'='100', 'numPartitions'='2' Two concurrent tasks are started in DLI. The execution ID of one task is greater than or equal to **0** and the ID is less than **50**, and the execution ID of the other task is greater than or equal to **50** and the ID is less than **100**.
fetchsize	Number of data records obtained in each batch during data reading. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
batchsize	Number of data records written in each batch. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied. If this parameter is set to a large value, memory overflow may occur.
truncate	Indicates whether to clear the table without deleting the original table when **overwrite** is executed. The options are as follows: ● true ● false The default value is **false**, indicating that the original table is deleted and then a new table is created when the **overwrite** operation is performed.
isolationLevel	Transaction isolation level. The options are as follows: ● NONE ● READ_UNCOMMITTED ● READ_COMMITTED ● REPEATABLE_READ ● SERIALIZABLE The default value is **READ_UNCOMMITTED**.

Figure 4-36 GaussDB(DWS) cluster information



2. Insert data

```
sparkSession.sql("insert into dli_to_dws values(1, 'John',24),(2, 'Bob',32)")
```

3. Query data

```
val dataFrame = sparkSession.sql("select * from dli_to_dws")
dataFrame.show()
```

Before data is inserted:

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  7|   chm| 13|
|  6|    qz| 13|
|  3|  mark| 20|
+---+------+---+
```

Response:

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  6|    qz| 13|
|  7|   chm| 13|
|  3|  mark| 20|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  1|  John| 24|
|  2|   Bob| 32|
+---+------+---+
```

4. Delete the datasource connection table.

```
sparkSession.sql("drop table dli_to_dws")
```


Accessing a Data Source Using a DataFrame API

1. Set connection parameters.

```
val url = "jdbc:postgresql://to-dws-1174405057-EA1Kgo8H.datasource.com:8000/postgres"
val username = "dbadmin"
val password = "######"
val dbtable = "customer"
```

2. Create a DataFrame, add data, and rename fields

```
var dataFrame_1 = sparkSession.createDataFrame(List((8, "Jack_1", 18)))
val df = dataFrame_1.withColumnRenamed("_1", "id")
                    .withColumnRenamed("_2", "name")
                    .withColumnRenamed("_3", "age")
```

3. Import data to GaussDB(DWS).

```
df.write.format("jdbc")
  .option("url", url)
  .option("dbtable", dbtable)
  .option("user", username)
  .option("password", password)
  .mode(SaveMode.Append)
  .save()
```

NOTE

The options of **SaveMode** can be one of the following:

- **ErrorIfExis**: If the data already exists, the system throws an exception.
- **Overwrite**: If the data already exists, the original data will be overwritten.
- **Append**: If the data already exists, the system saves the new data.
- **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

4. Read data from GaussDB(DWS).

   – Method 1: read.format()

```
val jdbcDF = sparkSession.read.format("jdbc")
                    .option("url", url)
                    .option("dbtable", dbtable)
                    .option("user", username)
                    .option("password", password)
                    .load()
```

   – Method 2: read.jdbc()

```
val properties = new Properties()
 properties.put("user", username)
 properties.put("password", password)
 val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)
```

Before data is inserted:

```
+---+------+---+\n
| id|  name|age|\n
+---+------+---+\n
|  4|  kobe| 24|\n
|  3|  mark| 20|\n
|  7|   chm| 13|\n
|  6|    qz| 13|\n
|  1|   tom| 18|\n
|  2|  ammy| 18|\n
|  5|jordan| 22|\n
+---+------+---+\n
```

Response:


```
+---+------+---+\n
| id|  name|age|\n
+---+------+---+\n
|  7|   chm| 13|\n
|  1|   tom| 18|\n
|  2|  ammy| 18|\n
|  5|jordan| 22|\n
|  8|Jack_1| 18|\n
|  3|  mark| 20|\n
|  6|    qz| 13|\n
|  4|  kobe| 24|\n
+---+------+---+\n
```

The dateFrame read by the **read.format()** or **read.jdbc()** method is registered as a temporary table. Then, you can use SQL statements to query data.

```
jdbcDF.registerTempTable("customer_test")
 sparkSession.sql("select * from customer_test where id = 1").show()
```

Query results

```
+---+----+---+
| id|name|age|
+---+----+---+
|  1| tom| 18|
+---+----+---+
```

DataFrame-Related Operations

The data created by the **createDataFrame()** method and the data queried by the **read.format()** method and the **read.jdbc()** method are all DataFrame objects. You can directly query a single record. (In **Accessing a Data Source Using a DataFrame API**, the DataFrame data is registered as a temporary table.)

- where

  The **where** statement can be used in conjunction with filter expressions like AND and OR. It returns the DataFrame object after applying the specified filters. Here is an example:

  ```
  jdbcDF.where("id = 1 or age <=10").show()
  ```

  ```
  +---+------+---+
  | id|  name|age|
  +---+------+---+
  |  7|   chm| 13|
  |  1|   tom| 18|
  |  2|  ammy| 18|
  |  5|jordan| 22|
  |  6|    qz| 13|
  |  3|  mark| 20|
  +---+------+---+
  ```

- filter

  The **filter** statement can be used in the same way as **where**. The DataFrame object after filtering is returned. The following is an example:

  ```
  jdbcDF.filter("id = 1 or age <=10").show()
  ```



```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  7|   chm| 13|
|  5|jordan| 22|
|  6|    qz| 13|
|  3|  mark| 20|
+---+------+---+
```

- select

  The **select** statement is used to query DataFrame objects of specific fields. It allows querying multiple fields at once. Here are some examples:

  - Example 1:
    jdbcDF.select("**id**").show()

    ```
    +---+
    | id|
    +---+
    |  4|
    |  7|
    |  3|
    |  6|
    |  1|
    |  2|
    |  5|
    +---+
    ```

  - Example 2:
    jdbcDF.select("**id**", "**name**").show()

    ```
    +---+------+
    | id|  name|
    +---+------+
    |  4|  kobe|
    |  7|   chm|
    |  6|    qz|
    |  3|  mark|
    |  1|   tom|
    |  2|  ammy|
    |  5|jordan|
    +---+------+
    ```

  - Example 3:
    jdbcDF.select("**id**","**name**").where("**id<4**").show()

    ```
    +---+----+
    | id|name|
    +---+----+
    |  1| tom|
    |  2|ammy|
    |  3|mark|
    +---+----+
    ```

- selectExpr

The **selectExpr** statement is used to perform special processing on a field. For example, it can be used to change the field name. The following is an example:

If you want to set the **name** field to **name_test** and add 1 to the value of **age**, run the following statement:

```
jdbcDF.selectExpr("id", "name as name_test", "age+1").show()
```

- col

  col is used to obtain a specified field. Different from **select**, **col** can only be used to query the column type and one field can be returned at a time. The following is an example:

  ```
  val idCol = jdbcDF.col("id")
  ```

- drop

  drop is used to delete a specified field. Specify a field you need to delete (only one field can be deleted at a time), the DataFrame object that does not contain the field is returned. Here is an example:

  ```
  jdbcDF.drop("id").show()
  ```

  ```
  +----+---+
  |name|age|
  +----+---+
  |  qz| 13|
  | chm| 13|
  | tom| 18|
  |ammy| 18|
  +----+---+
  ```

Submitting a Job

1. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.
2. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

   NOTE

   - For Spark 2.3.2 (soon to be take offline) or 2.4.5, set **Module** to **sys.datasource.dws** when submitting a job.
   - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

     spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*

     spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*
   - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
   - For how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Maven dependency

  ```
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
  ```



```
  <version>2.3.2</version>
</dependency>
```

- Connecting to data sources through SQL APIs

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

```
import java.util.Properties
import org.apache.spark.sql.SparkSession

object Test_SQL_DWS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    // Create a data table for DLI-associated DWS
    sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (
      'url'='jdbc:postgresql://to-dws-1174405057-EA1Kgo8H.datasource.com:8000/postgres',
      'dbtable'='customer',
      'user'='dbadmin',
      'password'='######')")

    //*****************************SQL model***********************************
    //Insert data into the DLI data table
    sparkSession.sql("insert into dli_to_dws values(1,'John',24),(2,'Bob',32)")

    //Read data from DLI data table
    val dataFrame = sparkSession.sql("select * from dli_to_dws")
    dataFrame.show()

    //drop table
    sparkSession.sql("drop table dli_to_dws")

    sparkSession.close()
  }
}
```

- Connecting to data sources through DataFrame APIs

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

```
import java.util.Properties
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.SaveMode

object Test_SQL_DWS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    //*****************************DataFrame model***********************************
    // Set the connection configuration parameters. Contains url, username, password, dbtable.
    val url = "jdbc:postgresql://to-dws-1174405057-EA1Kgo8H.datasource.com:8000/postgres"
    val username = "dbadmin"
    val password = "######"
    val dbtable = "customer"

    //Create a DataFrame and initialize the DataFrame data.
    var dataFrame_1 = sparkSession.createDataFrame(List((1, "Jack", 18)))

    //Rename the fields set by the createDataFrame() method.
    val df = dataFrame_1.withColumnRenamed("_1", "id")
                .withColumnRenamed("_2", "name")
                .withColumnRenamed("_3", "age")
```



```
  //Write data to the dws_table_1 table
  df.write.format("jdbc")
    .option("url", url)
    .option("dbtable", dbtable)
    .option("user", username)
    .option("password", password)
    .mode(SaveMode.Append)
    .save()

  // DataFrame object for data manipulation
  //Filter users with id=1
  var newDF = df.filter("id!=1")
  newDF.show()

  // Filter the id column data
  var newDF_1 = df.drop("id")
  newDF_1.show()

  // Read the data of the customer table in the RDS database
  //Way one: Read data from GaussDB(DWS) using read.format()
  val jdbcDF = sparkSession.read.format("jdbc")
                .option("url", url)
                .option("dbtable", dbtable)
                .option("user", username)
                .option("password", password)
                .option("driver", "org.postgresql.Driver")
                .load()
  //Way two: Read data from GaussDB(DWS) using read.jdbc()
  val properties = new Properties()
  properties.put("user", username)
  properties.put("password", password)
  val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)

  /**
   * Register the dateFrame read by read.format() or read.jdbc() as a temporary table, and query the
data
   * using the sql statement.
   */
  jdbcDF.registerTempTable("customer_test")
  val result = sparkSession.sql("select * from customer_test where id = 1")
  result.show()

  sparkSession.close()
 }
}
```

4.3.3.2 PySpark Example Code

Scenario

This section provides PySpark example code that demonstrates how to use a Spark job to access data from the GaussDB(DWS) data source.

A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

NOTE

Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

Preparations

1. Import dependency packages.
```
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession
```
2. Create a session.
```
sparkSession = SparkSession.builder.appName("datasource-dws").getOrCreate()
```

Accessing a Data Source Using a DataFrame API

1. Set connection parameters.
```
url = "jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres"
dbtable = "customer"
user = "dbadmin"
password = "######"
driver = "org.postgresql.Driver"
```
2. Set data.
```
dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19)])
```
3. Configure the schema.
```
schema = StructType([StructField("id", IntegerType(), False),\
                     StructField("name", StringType(), False),\
                     StructField("age", IntegerType(), False)])
```
4. Create a DataFrame.
```
dataFrame = sparkSession.createDataFrame(dataList, schema)
```
5. Save the data to GaussDB(DWS).
```
dataFrame.write \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .mode("Overwrite") \
    .save()
```

   NOTE

   The options of **mode** can be one of the following:

   - **ErrorIfExis**: If the data already exists, the system throws an exception.
   - **Overwrite**: If the data already exists, the original data will be overwritten.
   - **Append**: If the data already exists, the system saves the new data.
   - **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

6. Read data from GaussDB(DWS).
```
jdbcDF = sparkSession.read \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .load()
jdbcDF.show()
```
7. View the operation result.
```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  1|Katie| 19|
+---+-----+---+
```


Accessing a Data Source Using a SQL API

1. Create a table to connect to a GaussDB(DWS) data source.

```
sparkSession.sql(
    "CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (
    'url'='jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres',\
    'dbtable'='customer',\
    'user'='dbadmin',\
    'password'='######',\
    'driver'='org.postgresql.Driver')")
```

   NOTE

   For details about table creation parameters, see **Table 4-14**.

2. Insert data.

```
sparkSession.sql("insert into dli_to_dws values(2,'John',24)")
```

3. Query data.

```
jdbcDF = sparkSession.sql("select * from dli_to_dws").show()
```

4. View the operation result.

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  1|Katie| 19|
|  2| John| 24|
+---+-----+---+
```

Submitting a Spark Job

1. Upload the Python code file to the OBS bucket.
2. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

   NOTE

   - For Spark 2.3.2 (soon to be take offline) or 2.4.5, set **Module** to **sys.datasource.dws** when submitting a job.
   - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

     spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*

     spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*
   - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
   - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Connecting to data sources through DataFrame APIs

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
```



```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-dws").getOrCreate()

  # Set cross-source connection parameters
  url = "jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres"
  dbtable = "customer"
  user = "dbadmin"
  password = "######"
  driver = "org.postgresql.Driver"

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19)])

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),\
                       StructField("name", StringType(), False),\
                       StructField("age", IntegerType(), False)])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(dataList, schema)

  # Write data to the DWS table
  dataFrame.write \
     .format("jdbc") \
     .option("url", url) \
     .option("dbtable", dbtable) \
     .option("user", user) \
     .option("password", password) \
     .option("driver", driver) \
     .mode("Overwrite") \
     .save()

  # Read data
  jdbcDF = sparkSession.read \
     .format("jdbc") \
     .option("url", url) \
     .option("dbtable", dbtable) \
     .option("user", user) \
     .option("password", password) \
     .option("driver", driver) \
     .load()
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

- Connecting to data sources through SQL APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-dws").getOrCreate()

  # Create a data table for DLI - associated GaussDB(DWS)
  sparkSession.sql(
     "CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (\
     'url'='jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres',\
     'dbtable'='customer',\
     'user'='dbadmin',\
     'password'='######',\
     'driver'='org.postgresql.Driver')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into dli_to_dws values(2,'John',24)")
```



```
# Read data from DLI data table
jdbcDF = sparkSession.sql("select * from dli_to_dws").show()

# close session
sparkSession.stop()
```

4.3.3.3 Java Example Code

Scenario

This section provides Java example code that demonstrates how to use a Spark job to access data from the GaussDB(DWS) data source.

A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

NOTE

Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

Preparations

1. Import dependencies.
   - Maven dependency involved
     ```
     <dependency>
       <groupId>org.apache.spark</groupId>
       <artifactId>spark-sql_2.11</artifactId>
       <version>2.3.2</version>
     </dependency>
     ```
   - Import dependency packages.
     ```
     import org.apache.spark.sql.SparkSession;
     ```
2. Create a session.
   ```
   SparkSession sparkSession = SparkSession.builder().appName("datasource-dws").getOrCreate();
   ```

Accessing a Data Source Through a SQL API

1. Create a table to connect to a GaussDB(DWS) data source and set connection parameters.
   ```
   sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS
   ('url'='jdbc:postgresql://10.0.0.233:8000/postgres','dbtable'='test','user'='dbadmin','password'='**')");
   ```
2. Insert data.
   ```
   sparkSession.sql("insert into dli_to_dws values(3,'Liu'),(4,'Xie')");
   ```
3. Query data.
   ```
   sparkSession.sql("select * from dli_to_dws").show();
   ```

   Response:

   ```
   +----+-----------+
   |  id|       name|
   +----+-----------+
   |1111|gff_test001|
   |   3|        Liu|
   |   1|       John|
   |   2|        Bob|
   |   4|        Xie|
   +----+-----------+
   ```


Submitting a Spark Job

1. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.
2. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

   NOTE

   - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.dws** when you submit a job.
   - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

     spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*

     spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*
   - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
   - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

Accessing GaussDB(DWS) tables through SQL APIs

```
import org.apache.spark.sql.SparkSession;

public class java_dws {
    public static void main(String[] args) {
        SparkSession sparkSession = SparkSession.builder().appName("datasource-dws").getOrCreate();

        sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS
('url'='jdbc:postgresql://10.0.0.233:8000/postgres','dbtable'='test','user'='dbadmin','password'='**')");

        //*****************************SQL model***********************************
        //Insert data into the DLI data table
        sparkSession.sql("insert into dli_to_dws values(3,'Liu'),(4,'Xie')");

        //Read data from DLI data table
        sparkSession.sql("select * from dli_to_dws").show();

        //drop table
        sparkSession.sql("drop table dli_to_dws");

        sparkSession.close();
    }
}
```

4.3.4 Connecting to HBase

4.3.4.1 MRS Configuration

Configuring MRS Host Information in DLI Datasource Connection

1. Create a datasource connection on the DLI management console. For details, see **Data Lake Insight User Guide**.
2. Add the **/etc/hosts** information of MRS cluster nodes to the **host** file of the DLI queue.

For details, see **Modifying the Host Information** in the *Data Lake Insight User Guide*.

Completing Configurations for Enabling Kerberos Authentication

1. Create a cluster with Kerberos authentication enabled by referring to section "Creating a Security Cluster and Logging In to MRS Manager" in **Using Clusters with Kerberos Authentication Enabled**. Add a user and grant permissions to the user by referring to section "Creating Roles and Users".
2. Use the user created in **1** for login authentication. For details, see **Using an HBase Client**. A human-machine user must change the password upon the first login.
3. Log in to Manager and choose **System**. In the navigation pane on the left, choose **Permission** > **User**, locate the row where the new user locates, click **More**, and select **Download Authentication Credential**. Save the downloaded package and decompress it to obtain the **keytab** and **krb5.conf** files.

Creating an MRS HBase Table

Before creating an MRS HBase table to be associated with the DLI table, ensure that the HBase table exists. The following provides example code to describe how to create an MRS HBase table:

1. Remotely log in to the ECS and use the HBase Shell command to view table information. In this command, **hbtest** indicates the name of the table to be queried.
   ```
   describe 'hbtest'
   ```
2. (Optional) If the HBase table does not exist, run the following command to create one:
   ```
   create 'hbtest', 'info', 'detail'
   ```
   In this command, **hbtest** indicates the table name, and other parameters indicate the column family names.
3. Configure the connection information. **TableName** corresponds to the name of the HBase table.

4.3.4.2 Scala Example Code

Development Description

The CloudTable HBase and MRS HBase can be connected to DLI as data sources.

- Prerequisites

  A datasource connection has been created on the DLI management console. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Constructing dependency information and creating a Spark session

a. Import dependencies.

Maven dependency involved

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

Import dependency packages.

```
import scala.collection.mutable
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._
```

b. Create a session.

```
val sparkSession = SparkSession.builder().getOrCreate()
```

c. Create a table to connect to an HBase data source.

- The sample code is applicable, if Kerberos authentication **is disabled** for the interconnected HBase cluster:

```
sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING,
'booleanf' BOOLEAN,
       'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase
OPTIONS (
   'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
             cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
             cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
   'TableName'='table_DupRowkey1',
   'RowKey'='id:5,location:6,city:7',

'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,d
oublef:CF1.doublef')"
)
```

- The sample code is applicable, if Kerberos authentication **is enabled** for the interconnected HBase cluster:

```
sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING,
'booleanf' BOOLEAN,
       'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase
OPTIONS (
   'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
             cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
             cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
   'TableName'='table_DupRowkey1',
   'RowKey'='id:5,location:6,city:7',

'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,d
oublef:CF1.doublef',
   'krb5conf'='./krb5.conf',
   'keytab' = './user.keytab',
   'principal' = 'krbtest')")
```


Table 4-15 Parameters for creating a table

Parameter	Description
ZKHost	ZooKeeper IP address of the HBase cluster. You need to create a datasource connection first. For details, see **Enhanced Datasource Connections**. • To access the CloudTable cluster, specify the ZooKeeper connection address in the internal network. • To access the MRS cluster, specify the IP addresses and port numbers of the ZooKeeper nodes. The format is as follows: ZK_IP1:ZK_PORT1,ZK_IP2:ZK_PORT2
RowKey	Row key field of the table connected to DLI. The single and composite row keys are supported. A single row key can be of the numeric or string type. The length does not need to be specified. The composite row key supports only fixed-length data of the string type. The format is ***attribute name 1*:*Length*, *attribute name 2*:*Length***.
Cols	Mapping between the fields in the DLI table and the CloudTable table. In this mapping, the DLI table field is placed before the colon (:) and the CloudTable table field is placed after the colon (:). The period (.) is used to separate the column family and column name of the CloudTable table. For example: **DLI table field 1:CloudTable table.CloudTable table field 1**, **DLI table field 2:CloudTable table.CloudTable table field 2**, **DLI table field 3:CLoudTable table.CloudTable table field 3**
krb5conf	Path of the **krb5.conf** file. This parameter is required when Kerberos authentication is enabled. The format is './krb5.conf'. For details, see **Completing Configurations for Enabling Kerberos Authentication**.
keytab	Path of the **keytab** file. This parameter is required when Kerberos authentication is enabled. The format is './user.keytab.'. For details, see **Completing Configurations for Enabling Kerberos Authentication**.
principal	Username created for Kerberos authentication.

Accessing a Data Source Using a SQL API

1. Insert data.
   ```
   sparkSession.sql("insert into test_hbase values('12345','abc','guiyang',false,null,3,23,2.3,2.34)")
   ```
2. Query data.
   ```
   sparkSession.sql("select * from test_hbase").show ()
   ```
   Response

Accessing a Data Source Using a DataFrame API

1. Construct a schema.

```
val attrId = new StructField("id",StringType)
val location = new StructField("location",StringType)
val city = new StructField("city",StringType)
val booleanf = new StructField("booleanf",BooleanType)
val shortf = new StructField("shortf",ShortType)
val intf = new StructField("intf",IntegerType)
val longf = new StructField("longf",LongType)
val floatf = new StructField("floatf",FloatType)
val doublef = new StructField("doublef",DoubleType)
val attrs = Array(attrId, location,city,booleanf,shortf,intf,longf,floatf,doublef)
```

2. Construct data based on the schema type.

```
val mutableRow: Seq[Any] = Seq("12345","abc","city1",false,null,3,23,2.3,2.34)
val rddData: RDD[Row] = sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)), 1)
```

3. Import data to HBase.

```
sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("test_hbase")
```

4. Read data from HBase.

```
val map = new mutable.HashMap[String, String]()
map("TableName") = "table_DupRowkey1"
map("RowKey") = "id:5,location:6,city:7"
map("Cols") =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doubl
ef"
map("ZKHost")="cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
            cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
            cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181"
sparkSession.read.schema(new StructType(attrs)).format("hbase").options(map.toMap).load().show()
```

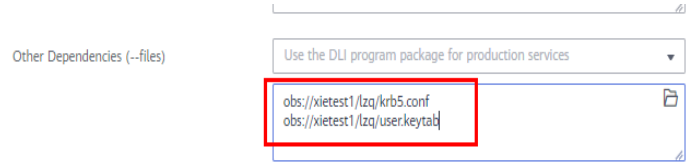
Returned result:

```
+-----+--------+-------+--------+------+----+-----+------+-------+\n
|   id|location|   city|booleanf|shortf|intf|longf|floatf|doublef|\n
+-----+--------+-------+--------+------+----+-----+------+-------+\n
|12345|  huawei|guiyang|   false|  null|   3|   23|   2.3|   2.34|\n
|abcde|  abcdef|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
|check|  abcdef|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
\n+-----+--------+-------+--------+------+----+-----+------+-------+\n\n
```

Submitting a Spark Job

1. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.
2. (Optional) Add the **krb5.conf** and **user.keytab** files to other dependency files of the job when creating a Spark job in an MRS cluster with Kerberos authentication enabled. Skip this step if Kerberos authentication is not enabled for the cluster.

Figure 4-37 Adding dependency files



3. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

NOTE

- If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, set **Module** to **sys.datasource.hbase** when you submit a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Maven dependency

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- Connecting to data sources through SQL APIs
  - Sample code when Kerberos authentication is disabled

```
import org.apache.spark.sql.SparkSession

object Test_SparkSql_HBase {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    /**
     * Create an association table for the DLI association Hbase table
     */
    sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING,
'booleanf' BOOLEAN,
        'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase
OPTIONS (
    'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
            cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
            cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
    'TableName'='table_DupRowkey1',
    'RowKey'='id:5,location:6,city:7',
    'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,
        longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doublef')")

    //*****************************SQL model***********************************
    sparkSession.sql("insert into test_hbase values('12345','abc','city1',false,null,3,23,2.3,2.34)")
    sparkSession.sql("select * from test_hbase").collect()

    sparkSession.close()
  }
}
```

  - Sample code when Kerberos authentication is enabled

```
import org.apache.spark.SparkFiles
import org.apache.spark.sql.SparkSession

import java.io.{File, FileInputStream, FileOutputStream}

object Test_SparkSql_HBase_Kerberos {

  def copyFile2(Input:String)(OutPut:String): Unit ={
```



```
    val fis = new FileInputStream(Input)
    val fos = new FileOutputStream(OutPut)
    val buf = new Array[Byte](1024)
    var len = 0
    while ({len = fis.read(buf);len} != -1){
      fos.write(buf,0,len)
    }
    fos.close()
    fis.close()
  }

  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    val sc = sparkSession.sparkContext
    sc.addFile("OBS address of krb5.conf")
    sc.addFile("OBS address of user.keytab")
    Thread.sleep(10)

    val krb5_startfile = new File(SparkFiles.get("krb5.conf"))
    val keytab_startfile = new File(SparkFiles.get("user.keytab"))
    val path_user = System.getProperty("user.dir")
    val keytab_endfile = new File(path_user + "/" + keytab_startfile.getName)
    val krb5_endfile = new File(path_user + "/" + krb5_startfile.getName)
    println(keytab_endfile)
    println(krb5_endfile)

    var krbinput = SparkFiles.get("krb5.conf")
    var krboutput = path_user+"/krb5.conf"
    copyFile2(krbinput)(krboutput)

    var keytabinput = SparkFiles.get("user.keytab")
    var keytaboutput = path_user+"/user.keytab"
    copyFile2(keytabinput)(keytaboutput)
    Thread.sleep(10)
    /**
     * Create an association table for the DLI association Hbase table
     */
    sparkSession.sql("CREATE TABLE testhbase(id string,booleanf boolean,shortf short,intf
int,longf long,floatf float,doublef double) " +
      "using hbase OPTIONS(" +
      "'ZKHost'='10.0.0.146:2181'," +
      "'TableName'='hbtest'," +
      "'RowKey'='id:100'," +

"'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF2.longf,floatf:CF1.floatf,doubl
ef:CF2.doublef'," +
      "'krb5conf'='" + path_user + "/krb5.conf'," +
      "'keytab'='" + path_user+ "/user.keytab'," +
      "'principal'='krbtest') ")

  //*****************************SQL model***********************************
  sparkSession.sql("insert into testhbase values('newtest',true,1,2,3,4,5)")
  val result = sparkSession.sql("select * from testhbase")
  result.show()

  sparkSession.close()
  }
}
```

- Connecting to data sources through DataFrame APIs

```
import scala.collection.mutable

import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._

object Test_SparkSql_HBase {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
```



```
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create an association table for the DLI association Hbase table
    sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING, 'booleanf'
BOOLEAN,
        'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase OPTIONS (
    'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
            cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
            cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
    'TableName'='table_DupRowkey1',
    'RowKey'='id:5,location:6,city:7',

'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.
doublef')")

    //*****************************DataFrame model***********************************
    // Setting schema
    val attrId = new StructField("id",StringType)
    val location = new StructField("location",StringType)
    val city = new StructField("city",StringType)
    val booleanf = new StructField("booleanf",BooleanType)
    val shortf = new StructField("shortf",ShortType)
    val intf = new StructField("intf",IntegerType)
    val longf = new StructField("longf",LongType)
    val floatf = new StructField("floatf",FloatType)
    val doublef = new StructField("doublef",DoubleType)
    val attrs = Array(attrId, location,city,booleanf,shortf,intf,longf,floatf,doublef)

    // Populate data according to the type of schema
    val mutableRow: Seq[Any] = Seq("12345","abc","city1",false,null,3,23,2.3,2.34)
    val rddData: RDD[Row] = sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)),
1)

    // Import the constructed data into Hbase
    sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("test_hbase")

    // Read data on Hbase
    val map = new mutable.HashMap[String, String]()
    map("TableName") = "table_DupRowkey1"
    map("RowKey") = "id:5,location:6,city:7"
    map("Cols") =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doubl
ef"
    map("ZKHost")="cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
                cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
                cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181"
    sparkSession.read.schema(new
StructType(attrs)).format("hbase").options(map.toMap).load().collect()

    sparkSession.close()
  }
}
```

4.3.4.3 PySpark Example Code

Development Description

The CloudTable HBase and MRS HBase can be connected to DLI as data sources.

- Prerequisites

  A datasource connection has been created on the DLI management console. For details, see **Enhanced Datasource Connections**.

NOTE

Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation
  a. Import dependency packages.
  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType, BooleanType, ShortType, LongType, FloatType, DoubleType
  from pyspark.sql import SparkSession
  ```
  b. Create a session.
  ```
  sparkSession = SparkSession.builder.appName("datasource-hbase").getOrCreate()
  ```
- Connecting to data sources through SQL APIs
  a. Create a table to connect to an HBase data source.
     - The sample code is applicable, if Kerberos authentication **is disabled** for the interconnected HBase cluster:
     ```
     sparkSession.sql(
         "CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase OPTIONS (\
         'ZKHost' = '192.168.0.189:2181',\
         'TableName' = 'hbtest',\
         'RowKey' = 'id:5',\
         'Cols' = 'location:info.location,city:detail.city')")
     ```
     - The sample code is applicable, if Kerberos authentication **is enabled** for the interconnected HBase cluster:
     ```
     sparkSession.sql(
         "CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase OPTIONS (\
         'ZKHost' = '192.168.0.189:2181',\
         'TableName' = 'hbtest',\
         'RowKey' = 'id:5',\
         'Cols' = 'location:info.location,city:detail.city',\
         'krb5conf' = './krb5.conf',\
         'keytab'='./user.keytab',\
         'principal' ='krbtest')")
     ```
     If Kerberos authentication is enabled, you need to set three more parameters, as listed in **Table 4-16**.

     Table 4-16 Description

Parameter and Value	Description
'krb5conf' = './krb5.conf'	Path of the **krb5.conf** file.
'keytab'='./user.keytab'	Path of the **keytab** file.
'principal' ='krbtest'	Authentication username.

     For details about how to obtain the **krb5.conf** and **keytab** files, see **Completing Configurations for Enabling Kerberos Authentication**.

NOTE

For details about parameters in the table, see **Table 4-15**.

b. Import data to HBase.

```
sparkSession.sql("insert into testhbase values('95274','abc','Jinan')")
```

c. Read data from HBase.

```
sparkSession.sql("select * from testhbase").show()
```

- Connecting to data sources through DataFrame APIs

a. Create a table to connect to an HBase data source.

```
sparkSession.sql(\
  "CREATE TABLE test_hbase(id STRING, location STRING, city STRING, booleanf BOOLEAN,
shortf SHORT, intf INT, longf LONG,
     floatf FLOAT, doublef DOUBLE) using hbase OPTIONS (\
  'ZKHost' = 'cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,\
              cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,\
              cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',\
  'TableName' = 'table_DupRowkey1',\
  'RowKey' = 'id:5,location:6,city:7',\
  'Cols' = 'booleanf:CF1.booleanf, shortf:CF1.shortf, intf:CF1.intf, \  longf:CF1.longf,
floatf:CF1.floatf, doublef:CF1.doublef')")
```

NOTE

- For details about the **ZKHost**, **RowKey**, and **Cols** parameters, see **Table 4-15**.
- **TableName**: Name of a table in the CloudTable file. If no table name exists, the system automatically creates one.

b. Construct a schema.

```
schema = StructType([StructField("id", StringType()),\
                     StructField("location", StringType()),\
                     StructField("city", StringType()),\
                     StructField("booleanf", BooleanType()),\
                     StructField("shortf", ShortType()),\
                     StructField("intf", IntegerType()),\
                     StructField("longf", LongType()),\
                     StructField("floatf", FloatType()),\
                     StructField("doublef", DoubleType())])
```

c. Set data.

```
dataList = sparkSession.sparkContext.parallelize([("11111", "aaa", "aaa", False, 4, 3, 23, 2.3,
2.34)])
```

d. Create a DataFrame.

```
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

e. Import data to HBase.

```
dataFrame.write.insertInto("test_hbase")
```

f. Read data from HBase.

```
// Set cross-source connection parameters
TableName = "table_DupRowkey1"
RowKey = "id:5,location:6,city:7"
Cols =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.
doublef"
ZKHost = "cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,cloudtable-cf82-zk2-
weBkIrjI.cloudtable.com:2181,
        cloudtable-cf82-zk1- WY09px9l.cloudtable.com:2181"

// select
jdbcDF = sparkSession.read.schema(schema)\
                .format("hbase")\
                .option("ZKHost",ZKHost)\
                .option("TableName",TableName)\
                .option("RowKey",RowKey)\
                .option("Cols",Cols)\
```



```
            .load()
jdbcDF.filter("id = '12333' or id='11111'").show()
```

NOTE

The length of **id**, **location**, and **city** parameter is limited. When inserting data, you must set the data values based on the required length. Otherwise, an encoding format error occurs during query.

g. View the operation result.

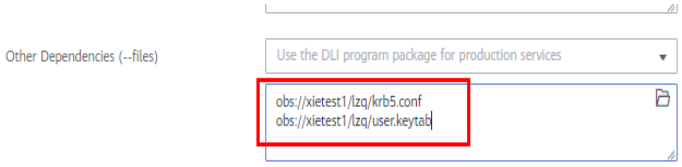
```
+-----+--------+-------+--------+------+----+-----+------+-------+
|   id|location|   city|booleanf|shortf|intf|longf|floatf|doublef|
+-----+--------+-------+--------+------+----+-----+------+-------+
|11111|  beijin|beijing|   false|     4|   3|   23|   2.3|   2.34|
|12333|  nanjin|nanjing|   false|  null|   3|   23|   2.3|   2.34|
+-----+--------+-------+--------+------+----+-----+------+-------+
```

- Submitting a Spark job

  a. Upload the Python code file to the OBS bucket.

  b. (Optional) Add the **krb5.conf** and **user.keytab** files to other dependency files of the job when creating a Spark job in an MRS cluster with Kerberos authentication enabled. Skip this step if Kerberos authentication is not enabled for the cluster. **Figure 4-38** shows how to add the files.

  Figure 4-38 Adding dependency files

  c. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  NOTE

  - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.hbase** when you submit a job.
  - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Connecting to MRS HBase through SQL APIs
  - Sample code when Kerberos authentication is **disabled**

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, BooleanType, ShortType, LongType, FloatType, DoubleType
```



```
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-hbase").getOrCreate()

  sparkSession.sql(
    "CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase OPTIONS (\
    'ZKHost' = '192.168.0.189:2181',\
    'TableName' = 'hbtest',\
    'RowKey' = 'id:5',\
    'Cols' = 'location:info.location,city:detail.city')")


  sparkSession.sql("insert into testhbase values('95274','abc','Jinan')")

  sparkSession.sql("select * from testhbase").show()
  # close session
  sparkSession.stop()
```

  - Sample code when Kerberos authentication is **enabled**

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark import SparkFiles
from pyspark.sql import SparkSession
import shutil
import time
import os

if __name__ == "__main__":
    # Create a SparkSession session.
    sparkSession =
SparkSession.builder.appName("Test_HBase_SparkSql_Kerberos").getOrCreate()
    sc = sparkSession.sparkContext
    time.sleep(10)

    krb5_startfile = SparkFiles.get("krb5.conf")
    keytab_startfile = SparkFiles.get("user.keytab")
    path_user = os.getcwd()
    krb5_endfile = path_user + "/" + "krb5.conf"
    keytab_endfile = path_user + "/" + "user.keytab"
    shutil.copy(krb5_startfile, krb5_endfile)
    shutil.copy(keytab_startfile, keytab_endfile)
    time.sleep(20)

    sparkSession.sql(
      "CREATE TABLE testhbase(id string,booleanf boolean,shortf short,intf int,longf long,floatf
float,doublef double) " +
      "using hbase OPTIONS(" +
      "'ZKHost'='10.0.0.146:2181'," +
      "'TableName'='hbtest'," +
      "'RowKey'='id:100'," +

"'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF2.longf,floatf:CF1.floatf,doubl
ef:CF2.doublef'," +
      "'krb5conf'='" + path_user + "/krb5.conf'," +
      "'keytab'='" + path_user+ "/user.keytab'," +
      "'principal'='krbtest') ")

      sparkSession.sql("insert into testhbase values('95274','abc','Jinan')")

    sparkSession.sql("select * from testhbase").show()
    # close session
    sparkSession.stop()
```

- Connecting to HBase through DataFrame APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, BooleanType,
ShortType, LongType, FloatType, DoubleType
```



```
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-hbase").getOrCreate()

  # Createa data table for DLI-associated ct
  sparkSession.sql(\
   "CREATE TABLE test_hbase(id STRING, location STRING, city STRING, booleanf BOOLEAN, shortf
SHORT, intf INT, longf LONG,floatf FLOAT,doublef DOUBLE) using hbase OPTIONS ( \
     'ZKHost' = 'cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,\
                 cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,\
                 cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',\
    'TableName' = 'table_DupRowkey1',\
    'RowKey' = 'id:5,location:6,city:7',\
    'Cols' =
'booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.double
f')")

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([("11111", "aaa", "aaa", False, 4, 3, 23, 2.3, 2.34)])

  # Setting schema
  schema = StructType([StructField("id", StringType()),
                       StructField("location", StringType()),
                       StructField("city", StringType()),
                       StructField("booleanf", BooleanType()),
                       StructField("shortf", ShortType()),
                       StructField("intf", IntegerType()),
                       StructField("longf", LongType()),
                       StructField("floatf", FloatType()),
                       StructField("doublef", DoubleType())])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(dataList, schema)

  # Write data to the cloudtable-hbase
  dataFrame.write.insertInto("test_hbase")

  # Set cross-source connection parameters
  TableName = "table_DupRowkey1"
  RowKey = "id:5,location:6,city:7"
  Cols =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doubl
ef"
  ZKHost = "cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,cloudtable-cf82-zk2-
weBkIrjI.cloudtable.com:2181,
           cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181"
  # Read data on CloudTable-HBase
  jdbcDF = sparkSession.read.schema(schema)\
                       .format("hbase")\
                       .option("ZKHost", ZKHost)\
                       .option("TableName",TableName)\
                       .option("RowKey", RowKey)\
                       .option("Cols", Cols)\
                       .load()
  jdbcDF.filter("id = '12333' or id='11111'").show()

  # close session
  sparkSession.stop()
```

4.3.4.4 Java Example Code

Development Description

This example applies only to MRS HBase.

- Prerequisites

  A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation

  a. Import dependencies.

  - Maven dependency involved

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

  - Import dependency packages.

    ```
    import org.apache.spark.sql.SparkSession;
    ```

  b. Create a session.

  ```
  parkSession = SparkSession.builder().appName("datasource-HBase-MRS").getOrCreate();
  ```

- Connecting to data sources through SQL APIs

  – For clusters with Kerberos authentication disabled

  i. Create a table to connect to an MRS HBase data source and set connection parameters.

  ```
  sparkSession.sql("CREATE TABLE testhbase(id STRING, location STRING, city STRING)
  using hbase
  OPTIONS('ZKHost'='10.0.0.63:2181','TableName'='hbtest','RowKey'='id:5','Cols'='location:inf
  o.location,city:detail.city') ");
  ```

  ii. Insert data.

  ```
  sparkSession.sql("insert into testhbase values('12345','abc','xxx')");
  ```

  iii. Query data.

  ```
  sparkSession.sql("select * from testhbase").show();
  ```

  Response

  ```
  +-----+--------+--------+
  |   id|location|    city|
  +-----+--------+--------+
  |12342|Shanghai|Shanghai|
  |12345|     abc|  guiyang|
  +-----+--------+--------+
  ```

  – For clusters with Kerberos authentication enabled

  i. Create a table to connect to an MRS HBase data source and set connection parameters.

  ```
  sparkSession.sql("CREATE TABLE testhbase(id STRING, location STRING, city STRING)
  using hbase
  OPTIONS('ZKHost'='10.0.0.63:2181','TableName'='hbtest','RowKey'='id:5','Cols'='location:inf
  o.location,city:detail.city,'krb5conf'='./krb5.conf','keytab'='./
  user.keytab','principal'='krbtest') ");
  ```

  If Kerberos authentication is enabled, you need to set three more parameters, as listed in **Table 4-17**.

Table 4-17 Parameter description

Parameter and Value	Description
'krb5conf' = './krb5.conf'	Path of the **krb5.conf** file.
'keytab'='./user.keytab'	Path of the **keytab** file.
'principal' ='krbtest'	Authentication username.

For details about how to obtain the **krb5.conf** and **keytab** files, see **Completing Configurations for Enabling Kerberos Authentication**.

ii. Insert data.

```
sparkSession.sql("insert into testhbase values('95274','abc','Hongkong')");
```

iii. Query data.

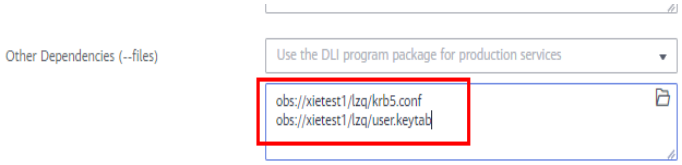
```
sparkSession.sql("select * from testhbase").show();
```

- Submitting a Spark job

a. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

b. (Optional) Add the **krb5.conf** and **user.keytab** files to the dependency files of the job when creating a Spark job in an MRS cluster with Kerberos authentication enabled. Skip this step if Kerberos authentication is not enabled for the cluster. **Figure 4-39** shows how to add the files.

Figure 4-39 Adding dependency files



c. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

NOTE

- If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.hbase** when you submit a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Connecting to data sources through SQL APIs
  - Complete example code for the cluster with Kerberos authentication **disabled**

```
import org.apache.spark.sql.SparkSession;

public class java_mrs_hbase {

    public static void main(String[] args) {
        //create a SparkSession session
        SparkSession sparkSession = SparkSession.builder().appName("datasource-HBase-MRS").getOrCreate();

        sparkSession.sql("CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase
OPTIONS('ZKHost'='10.0.0.63:2181','TableName'='hbtest','RowKey'='id:5','Cols'='location:info.location,city:detail.city') ");

        //****************************SQL model**********************************
        sparkSession.sql("insert into testhbase values('95274','abc','Hongkong')");
        sparkSession.sql("select * from testhbase").show();

        sparkSession.close();
    }
}
```

  - Complete example code for the cluster with Kerberos authentication **enabled**

```
import org.apache.spark.SparkContext;
import org.apache.spark.SparkFiles;
import org.apache.spark.sql.SparkSession;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Test_HBase_SparkSql_Kerberos {

    private static void copyFile(File src,File dst) throws IOException {
        InputStream input  = null;
        OutputStream output = null;
        try {
            input = new FileInputStream(src);
            output = new FileOutputStream(dst);
            byte[] buf = new byte[1024];
```



```
            int bytesRead;
            while ((bytesRead = input.read(buf)) > 0) {
                output.write(buf, 0, bytesRead);
            }
        } finally {
            input.close();
            output.close();
        }
    }

    public static void main(String[] args) throws InterruptedException, IOException {
        SparkSession sparkSession =
SparkSession.builder().appName("Test_HBase_SparkSql_Kerberos").getOrCreate();
        SparkContext sc = sparkSession.sparkContext();
        sc.addFile("obs://xietest1/lzq/krb5.conf");
        sc.addFile("obs://xietest1/lzq/user.keytab");
        Thread.sleep(20);

        File krb5_startfile = new File(SparkFiles.get("krb5.conf"));
        File keytab_startfile = new File(SparkFiles.get("user.keytab"));
        String path_user = System.getProperty("user.dir");
        File keytab_endfile = new File(path_user + "/" + keytab_startfile.getName());
        File krb5_endfile = new File(path_user + "/" + krb5_startfile.getName());
        copyFile(krb5_startfile,krb5_endfile);
        copyFile(keytab_startfile,keytab_endfile);
        Thread.sleep(20);

        /**
         * Create an association table for the DLI association Hbase table
         */
        sparkSession.sql("CREATE TABLE testhbase(id string,booleanf boolean,shortf short,intf
int,longf long,floatf float,doublef double) " +
                "using hbase OPTIONS(" +
                "'ZKHost'='10.0.0.146:2181'," +
                "'TableName'='hbtest'," +
                "'RowKey'='id:100'," +

"'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF2.longf,floatf:CF1.floatf,doubl
ef:CF2.doublef'," +
                "'krb5conf'='" + path_user + "/krb5.conf'," +
                "'keytab'='" + path_user+ "/user.keytab'," +
                "'principal'='krbtest') ");

        //*****************************SQL model***********************************
        sparkSession.sql("insert into testhbase values('newtest',true,1,2,3,4,5)");
        sparkSession.sql("select * from testhbase").show();
        sparkSession.close();
    }
}
```

4.3.4.5 Troubleshooting

Problem 1

- Symptom

  The Spark job fails to be executed, and the job log indicates that the Java server connection or container fails to be started.

- Solution

  Check whether the host information of the datasource connection has been modified. If not, modify the host information by referring to **Configuring MRS Host Information in DLI Datasource Connection**. Then create and submit a Spark job again.

Problem 2

- Symptom

  A Spark job fails to be executed and "KrbException: Message stream modified (41)" is displayed in the job log.

- Solution

  Delete all **renew_lifetime = xxx** configurations from the **krb5.conf** configuration file. Then create and submit a Spark job again.

4.3.5 Connecting to OpenTSDB

4.3.5.1 Scala Example Code

Development Description

The CloudTable OpenTSDB and MRS OpenTSDB can be connected to DLI as data sources.

- Prerequisites

  A datasource connection has been created on the DLI management console. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Constructing dependency information and creating a Spark session

  a. Import dependencies.

  Maven dependency involved

  ```
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.3.2</version>
  </dependency>
  ```

  Import dependency packages.

  ```
  import scala.collection.mutable
  import org.apache.spark.sql.{Row, SparkSession}
  import org.apache.spark.rdd.RDD
  import org.apache.spark.sql.types._
  ```

  b. Create a session.

  ```
  val sparkSession = SparkSession.builder().getOrCreate()
  ```

  c. Create a table to connect to an OpenTSDB data source.

  ```
  sparkSession.sql("create table opentsdb_test using opentsdb options(
      'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
          'metric'='ctopentsdb',
      'tags'='city,location')")
  ```


Table 4-18 Parameters for creating a table

Parameter	Description
host	OpenTSDB IP address. • To access CloudTable OpenTSDB, specify the OpenTSDB connection address. You can log in to the CloudTable console, choose **Cluster Mode** and click the target cluster name, and obtain the OpenTSDB connection address from the cluster information. • You can also access the MRS OpenTSDB. If you have created an enhanced datasource connection, enter the IP address and port number of the node where the OpenTSDB is located. The format is **IP:PORT**. If the OpenTSDB has multiple nodes, separate their IP addresses by semicolons (;). For details about how to obtain the IP address, see *MRS cluster OpenTSDB IP address* and *MRS cluster OpenTSDB port number*. If you use a basic datasource connection, enter the connection address returned. For details about operations on the management console, see the *Data Lake Insight User Guide*.
metric	Name of the metric in OpenTSDB corresponding to the DLI table to be created.
tags	Tags corresponding to the metric, used for operations such as classification, filtering, and quick search. A maximum of 8 tags, including all **tagk** values under the metric, can be added and are separated by commas (,).

- Connecting to data sources through SQL APIs

  a. Insert data.

  ```
  sparkSession.sql("insert into opentsdb_test values('futian', 'abc', '1970-01-02 18:17:36', 30.0)")
  ```

  b. Query data.

  ```
  sparkSession.sql("select * from opentsdb_test").show()
  ```

  Response

  ```
  +-------+--------+-------------------+-----+\n
  |   city|location|          timestamp|value|\n
  +-------+--------+-------------------+-----+\n
  | futian|  huawei|1970-01-02 18:17:36| 30.0|\n
  |beijing|  huawei|1970-01-02 18:17:36| 30.0|\n
  +-------+--------+-------------------+-----+\n\n
  ```

- Connecting to data sources through DataFrame APIs

  a. Construct a schema.

  ```
  val attrTag1Location = new StructField("location", StringType)
  val attrTag2Name = new StructField("name", StringType)
  val attrTimestamp = new StructField("timestamp", LongType)
  val attrValue = new StructField("value", DoubleType)
  val attrs = Array(attrTag1Location, attrTag2Name, attrTimestamp, attrValue)
  ```

  b. Construct data based on the schema type.


```
val mutableRow: Seq[Any] = Seq("aaa", "abc", 123456L, 30.0)
val rddData: RDD[Row] =
sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)), 1)
```

c. Import data to OpenTSDB.

```
sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("opentsdb_test")
```

d. Read data from OpenTSDB.

```
val map = new mutable.HashMap[String, String]()
map("metric") = "ctopentsdb"
map("tags") = "city,location"
map("Host") = "opentsdb-3xcl8dir15m58z3.cloudtable.com:4242"
sparkSession.read.format("opentsdb").options(map.toMap).load().show()
```

Response

```
+-------+--------+-------------------+-----+\n
|   city|location|          timestamp|value|\n
+-------+--------+-------------------+-----+\n
| futian|  huawei|1970-01-02 18:17:36| 30.0|\n
|beijing|  huawei|1970-01-02 18:17:36| 30.0|\n
+-------+--------+-------------------+-----+\n\n
```

- Submitting a Spark job

  a. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  NOTE

  - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.opentsdb** when you submit a job.
  - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Maven dependency

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.SparkSession

object Test_OpenTSDB_CT {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a data table for DLI association OpenTSDB
```



```
    sparkSession.sql("create table opentsdb_test using opentsdb options(
    'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
    'metric'='ctopentsdb',
    'tags'='city,location')")

    //*****************************SQL module***********************************
    sparkSession.sql("insert into opentsdb_test values('futian', 'abc', '1970-01-02 18:17:36', 30.0)")
    sparkSession.sql("select * from opentsdb_test").show()

    sparkSession.close()
  }
}
```

- Connecting to data sources through DataFrame APIs

```
import scala.collection.mutable
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._

object Test_OpenTSDB_CT {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a data table for DLI association OpenTSDB
    sparkSession.sql("create table opentsdb_test using opentsdb options(
    'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
    'metric'='ctopentsdb',
    'tags'='city,location')")

    //*****************************DataFrame model***********************************
    // Setting schema
    val attrTag1Location = new StructField("location", StringType)
    val attrTag2Name = new StructField("name", StringType)
    val attrTimestamp = new StructField("timestamp", LongType)
    val attrValue = new StructField("value", DoubleType)
    val attrs = Array(attrTag1Location, attrTag2Name, attrTimestamp,attrValue)

    // Populate data according to the type of schema
    val mutableRow: Seq[Any] = Seq("aaa", "abc", 123456L, 30.0)
    val rddData: RDD[Row] = sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)),
1)

    //Import the constructed data into OpenTSDB
    sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("opentsdb_test")

    //Read data on OpenTSDB
    val map = new mutable.HashMap[String, String]()
    map("metric") = "ctopentsdb"
    map("tags") = "city,location"
    map("Host") = "opentsdb-3xcl8dir15m58z3.cloudtable.com:4242"
    sparkSession.read.format("opentsdb").options(map.toMap).load().show()

    sparkSession.close()
  }
}
```

4.3.5.2 PySpark Example Code

Development Description

The CloudTable OpenTSDB and MRS OpenTSDB can be connected to DLI as data sources.

- Prerequisites

  A datasource connection has been created on the DLI management console. For details, see **Enhanced Datasource Connections**.

> **NOTE**
>
> Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation

  a. Import dependency packages.

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, StringType, LongType, DoubleType
  from pyspark.sql import SparkSession
  ```

  b. Create a session.

  ```
  sparkSession = SparkSession.builder.appName("datasource-opentsdb").getOrCreate()
  ```

  c. Create a table to connect to an OpenTSDB data source.

  ```
  sparkSession.sql("create table opentsdb_test using opentsdb options(
    'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
    'metric'='ct_opentsdb',
    'tags'='city,location')")
  ```

  > **NOTE**
  >
  > For details about the **Host**, **metric**, and **tags** parameters, see **Table 4-18**.

- Connecting to data sources through SQL APIs

  a. Insert data.

  ```
  sparkSession.sql("insert into opentsdb_test values('aaa', 'abc', '2021-06-30 18:00:00', 30.0)")
  ```

  b. Query data.

  ```
  result = sparkSession.sql("SELECT * FROM opentsdb_test")
  ```

- Connecting to data sources through DataFrame APIs

  a. Construct a schema.

  ```
  schema = StructType([StructField("location", StringType()),\
                       StructField("name", StringType()), \
                       StructField("timestamp", LongType()),\
                       StructField("value", DoubleType())])
  ```

  b. Set data.

  ```
  dataList = sparkSession.sparkContext.parallelize([("aaa", "abc", 123456L, 30.0)])
  ```

  c. Create a DataFrame.

  ```
  dataFrame = sparkSession.createDataFrame(dataList, schema)
  ```

  d. Import data to OpenTSDB.

  ```
  dataFrame.write.insertInto("opentsdb_test")
  ```

  e. Read data from OpenTSDB.

  ```
  jdbdDF = sparkSession.read
     .format("opentsdb")\
     .option("Host","opentsdb-3xcl8dir15m58z3.cloudtable.com:4242")\
     .option("metric","ctopentsdb")\
     .option("tags","city,location")\
     .load()
  jdbdDF.show()
  ```

  f. View the operation result.

  ```
  +-------+--------+-------------------+-----+
  |   city|location|          timestamp|value|
  +-------+--------+-------------------+-----+
  | futian|  huawei|1970-01-02 18:17:36| 30.0|
  |nanjing|     tom|2019-08-28 00:00:00| 30.0|
  |beijing|  huawei|1970-01-02 18:17:36| 30.0|
  +-------+--------+-------------------+-----+
  ```


- Submitting a Spark job
  a. Upload the Python code file to the OBS bucket.
  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

     NOTE

     - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.opentsdb** when you submit a job.
     - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

       spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*

       spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*
     - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
     - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Connecting to MRS OpenTSDB through SQL APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, StringType, LongType, DoubleType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-opentsdb").getOrCreate()


  # Create a DLI cross-source association opentsdb data table
  sparkSession.sql(\
    "create table opentsdb_test using opentsdb options(\
    'Host'='10.0.0.171:4242',\
    'metric'='cts_opentsdb',\
    'tags'='city,location')")

  sparkSession.sql("insert into opentsdb_test values('aaa', 'abc', '2021-06-30 18:00:00', 30.0)")

  result = sparkSession.sql("SELECT * FROM opentsdb_test")
  result.show()

  # close session
  sparkSession.stop()
```

- Connecting to OpenTSDB through DataFrame APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, StringType, LongType, DoubleType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-opentsdb").getOrCreate()

  # Create a DLI cross-source association opentsdb data table
  sparkSession.sql(
    "create table opentsdb_test using opentsdb options(\
    'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',\
```



```
  'metric'='ct_opentsdb',\
  'tags'='city,location')")

 # Create a DataFrame and initialize the DataFrame data.
 dataList = sparkSession.sparkContext.parallelize([("aaa", "abc", 123456L, 30.0)])

 # Setting schema
 schema = StructType([StructField("location", StringType()),\
                      StructField("name", StringType()),\
                      StructField("timestamp", LongType()),\
                      StructField("value", DoubleType())])

 # Create a DataFrame from RDD and schema
 dataFrame = sparkSession.createDataFrame(dataList, schema)

 # Set cross-source connection parameters
 metric = "ctopentsdb"
 tags = "city,location"
 Host = "opentsdb-3xcl8dir15m58z3.cloudtable.com:4242"

 # Write data to the cloudtable-opentsdb
 dataFrame.write.insertInto("opentsdb_test")
 # ******* Opentsdb does not currently implement the ctas method to save data, so the save() method
cannot be used.*******
  # dataFrame.write.format("opentsdb").option("Host", Host).option("metric", metric).option("tags",
tags).mode("Overwrite").save()

 # Read data on CloudTable-OpenTSDB
 jdbdDF = sparkSession.read\
     .format("opentsdb")\
     .option("Host",Host)\
     .option("metric",metric)\
     .option("tags",tags)\
     .load()
 jdbdDF.show()

 # close session
 sparkSession.stop()
```

4.3.5.3 Java Example Code

Development Description

This example applies only to MRS OpenTSDB.

- Prerequisites

  A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation

  a. Import dependencies.

  - Maven dependency involved

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```


  - Import dependency packages.
    ```
    import org.apache.spark.sql.SparkSession;
    ```

b. Create a session.
```
sparkSession = SparkSession.builder().appName("datasource-opentsdb").getOrCreate();
```

- Connecting to data sources through SQL APIs
  - Create a table to connect to an MRS OpenTSDB data source and set connection parameters.
    ```
    sparkSession.sql("create table opentsdb_new_test using opentsdb
    options('Host'='10.0.0.171:4242','metric'='ctopentsdb','tags'='city,location')");
    ```

    NOTE

    For details about the **Host**, **metric**, and **tags** parameters, see **Table 4-18**.
  - Insert data.
    ```
    sparkSession.sql("insert into opentsdb_new_test values('Penglai', 'abc', '2021-06-30 18:00:00',
    30.0)");
    ```
  - Query data.
    ```
    sparkSession.sql("select * from opentsdb_new_test").show();
    ```

    Response
    ```
    +-------+--------+-------------------+-----+
    |   city|location|          timestamp|value|
    +-------+--------+-------------------+-----+
    |Penglai|     abc|2021-06-30 18:00:00| 30.0|
    +-------+--------+-------------------+-----+
    ```

- Submitting a Spark job

a. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

NOTE

- If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.opentsdb** when you submit a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Maven dependency involved
  ```
  <dependency>
          <groupId>org.apache.spark</groupId>
          <artifactId>spark-sql_2.11</artifactId>
          <version>2.3.2</version>
  </dependency>
  ```


- Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.SparkSession;

public class java_mrs_opentsdb {

    private static SparkSession sparkSession = null;

    public static void main(String[] args) {
        //create a SparkSession session
        sparkSession = SparkSession.builder().appName("datasource-opentsdb").getOrCreate();

        sparkSession.sql("create table opentsdb_new_test using opentsdb
options('Host'='10.0.0.171:4242','metric'='ctopentsdb','tags'='city,location')");

        //****************************SQL module*********************************
        sparkSession.sql("insert into opentsdb_new_test values('Penglai', 'abc', '2021-06-30 18:00:00',
30.0)");
        System.out.println("Penglai new timestamp");
        sparkSession.sql("select * from opentsdb_new_test").show();

       sparkSession.close();

    }
}
```

4.3.5.4 Troubleshooting

A Spark Job Fails to Be Executed and "No respond" Is Displayed in the Job Log

- Symptom

  A Spark job fails to be executed and "No respond" is displayed in the job log.

- Solution

  Create a Spark job again. When creating the job, add the **spark.sql.mrs.opentsdb.ssl.enabled=true** configuration item to the **Spark parameters (--conf)**.

4.3.6 Connecting to RDS

4.3.6.1 Scala Example Code

Development Description

- Prerequisites

  A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Constructing dependency information and creating a Spark session

  a. Import dependencies.

     Maven dependency involved


```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

Import dependency packages.

```
import java.util.Properties
import org.apache.spark.sql.{Row,SparkSession}
import org.apache.spark.sql.SaveMode
```

b. Create a session.

```
val sparkSession = SparkSession.builder().getOrCreate()
```

- Connecting to data sources through SQL APIs

a. Create a table to connect to an RDS data source and set connection parameters.

```
sparkSession.sql(
  "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (
     'url'='jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306',  // Set this parameter to the actual URL.
     'dbtable'='test.customer',
     'user'='root',  // Set this parameter to the actual user.
     'password'='######',  // Set this parameter to the actual password.
     'driver'='com.mysql.jdbc.Driver')")
```

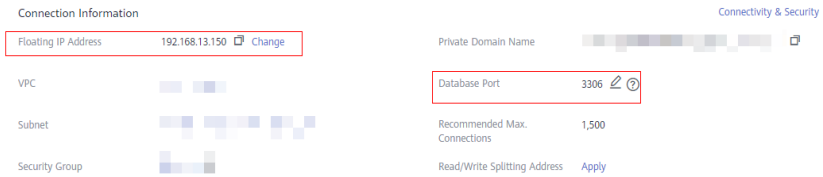
Table 4-19 Parameters for creating a table

Parameter	Description
url	To obtain an RDS IP address, you need to create a datasource connection first. Refer to the *Data Lake Insight User Guide* for more information. If you have created an enhanced datasource connection, use the internal network domain name or internal network address and the database port number provided by RDS to set up the connection. If MySQL is used, the format is ***Protocol header*://*Internal IP address*:*Internal network port number***. If PostgreSQL is used, the format is ***Protocol header*://*Internal IP address*:*Internal network port number*/*Database name***. For example: **jdbc:mysql://192.168.0.193:3306** or **jdbc:postgresql://192.168.0.193:3306/postgres**. For details about how to obtain the value, see *RDS cluster information*.
dbtable	To connect to a MySQL cluster, enter **Database name.Table name**. To connect to a PostgreSQL cluster, enter **Mode name.Table name**. **NOTE** If the database and table do not exist, create them first. Otherwise, the system reports an error and fails to run.
user	RDS database username.
password	RDS database password.

Parameter	Description
driver	JDBC driver class name. To connect to a MySQL cluster, enter **com.mysql.jdbc.Driver**. To connect to a PostgreSQL cluster, enter **org.postgresql.Driver**.
partitionColumn	One of the numeric fields that are required for concurrently reading data. **NOTE** • The **partitionColumn**, **lowerBound**, **upperBound**, and **numPartitions** parameters must be set at the same time. • To improve the concurrent read performance, you are advised to use auto-increment columns.
lowerBound	Minimum value of a column specified by **partitionColumn**. The value is contained in the returned result.
upperBound	Maximum value of a column specified by **partitionColumn**. The value is not contained in the returned result.
numPartitions	Number of concurrent read operations. **NOTE** When data is read, **lowerBound** and **upperBound** are evenly allocated to each task to obtain data. Example: 'partitionColumn'='id', 'lowerBound'='0', 'upperBound'='100', 'numPartitions'='2' Two concurrent tasks are started in DLI. The execution ID of one task is greater than or equal to **0** and the ID is smaller than **50**; the execution ID of the other task is greater than or equal to **50** and the ID is smaller than **100**.
fetchsize	Number of data records obtained in each batch during data reading. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied, causing memory overflow as a result.
batchsize	Number of data records written in each batch. The default value is **1000**. If this parameter is set to a large value, the performance is good but more memory is occupied, causing memory overflow as a result.

Parameter	Description
truncate	Whether to clear the table without deleting the original table when **overwrite** is executed. The options are as follows: • true • false The default value is **false**, indicating that the original table is deleted and then a new table is created when the **overwrite** operation is performed.
isolationLevel	Transaction isolation level. The options are as follows: • NONE • READ_UNCOMMITTED • READ_COMMITTED • REPEATABLE_READ • SERIALIZABLE The default value is **READ_UNCOMMITTED**.

Figure 4-40 RDS cluster information



b. Insert data.

```
sparkSession.sql("insert into dli_to_rds values(1, 'John',24),(2, 'Bob',32)")
```

c. Query data.

```
val dataFrame = sparkSession.sql("select * from dli_to_rds")
dataFrame.show()
```

Before data is inserted

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  7|   chm| 13|
|  6|    qz| 13|
|  3|  mark| 20|
+---+------+---+
```

After data is inserted


```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  6|    qz| 13|
|  7|   chm| 13|
|  3|  mark| 20|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  1|  John| 24|
|  2|   Bob| 32|
+---+------+---+
```

d. Delete the datasource connection table.

```
sparkSession.sql("drop table dli_to_rds")
```

- Connecting to data sources through DataFrame APIs

a. Configure datasource connection parameters.

```
val url = "jdbc:mysql://to-rds-1174405057-EA1Kgo8H.datasource.com:3306"
val username = "root"
val password = "######"
val dbtable = "test.customer"
```

b. Create a DataFrame, add data, and rename fields.

```
var dataFrame_1 = sparkSession.createDataFrame(List((8, "Jack_1", 18)))
val df = dataFrame_1.withColumnRenamed("_1", "id")
                    .withColumnRenamed("_2", "name")
                    .withColumnRenamed("_3", "age")
```

c. Import data to RDS.

```
df.write.format("jdbc")
  .option("url", url)
  .option("dbtable", dbtable)
  .option("user", username)
  .option("password", password)
  .option("driver", "com.mysql.jdbc.Driver")
  .mode(SaveMode.Append)
  .save()
```

NOTE

The value of **SaveMode** can be one of the following:

- **ErrorIfExis**: If the data already exists, the system throws an exception.
- **Overwrite**: If the data already exists, the original data will be overwritten.
- **Append**: If the data already exists, the system saves the new data.
- **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

d. Read data from RDS.

- Method 1: read.format()

```
val jdbcDF = sparkSession.read.format("jdbc")
                    .option("url", url)
                    .option("dbtable", dbtable)
                    .option("user", username)
                    .option("password", password)
                    .option("driver", "org.postgresql.Driver")
                    .load()
```

- Method 2: read.jdbc()

```
val properties = new Properties()
properties.put("user", username)
```



```
properties.put("password", password)
val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)
```

Before data is inserted

```
+---+------+---+\n
| id|  name|age|\n
+---+------+---+\n
|  4|  kobe| 24|\n
|  3|  mark| 20|\n
|  7|   chm| 13|\n
|  6|    qz| 13|\n
|  1|   tom| 18|\n
|  2|  ammy| 18|\n
|  5|jordan| 22|\n
+---+------+---+\n
```

After data is inserted

```
+---+------+---+\n
| id|  name|age|\n
+---+------+---+\n
|  7|   chm| 13|\n
|  1|   tom| 18|\n
|  2|  ammy| 18|\n
|  5|jordan| 22|\n
|  8|Jack_1| 18|\n
|  3|  mark| 20|\n
|  6|    qz| 13|\n
|  4|  kobe| 24|\n
+---+------+---+\n
```

The DataFrame read by the **read.format()** or **read.jdbc()** method is registered as a temporary table. Then, you can use SQL statements to query data.

```
jdbcDF.registerTempTable("customer_test")
sparkSession.sql("select * from customer_test where id = 1").show()
```

Query results

```
+---+----+---+
| id|name|age|
+---+----+---+
|  1| tom| 18|
+---+----+---+
```

- DataFrame-related operations

  The data created by the **createDataFrame()** method and the data queried by the **read.format()** method and the **read.jdbc()** method are all DataFrame objects. You can directly query a single record. (In **d**, the DataFrame data is registered as a temporary table.)

  – where

    The **where** statement can be used in conjunction with filter expressions like AND and OR. It returns the DataFrame object after applying the specified filters. Here is an example:

    ```
    jdbcDF.where("id = 1 or age <=10").show()
    ```



```
+---+------+---+
| id|  name|age|
+---+------+---+
|  7|   chm| 13|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  6|    qz| 13|
|  3|  mark| 20|
+---+------+---+
```

- filter

  The **filter** statement can be used in the same way as **where**. The DataFrame object after filtering is returned. The following is an example:

  ```
  jdbcDF.filter("id = 1 or age <=10").show()
  ```

  ```
  +---+------+---+
  | id|  name|age|
  +---+------+---+
  |  4|  kobe| 24|
  |  7|   chm| 13|
  |  5|jordan| 22|
  |  6|    qz| 13|
  |  3|  mark| 20|
  +---+------+---+
  ```

- select

  The **select** statement is used to query DataFrame objects of specific fields. It allows querying multiple fields at once. Here are some examples:

  - Example 1:

    ```
    jdbcDF.select("id").show()
    ```

    ```
    +---+
    | id|
    +---+
    |  4|
    |  7|
    |  3|
    |  6|
    |  1|
    |  2|
    |  5|
    +---+
    ```

  - Example 2:

    ```
    jdbcDF.select("id", "name").show()
    ```



```
+---+------+
| id|  name|
+---+------+
|  4|  kobe|
|  7|   chm|
|  6|    qz|
|  3|  mark|
|  1|   tom|
|  2|  ammy|
|  5|jordan|
+---+------+
```

- Example 3:
  jdbcDF.select(**"id"**,**"name"**).where(**"id<4"**).show()

  ```
  +---+----+
  | id|name|
  +---+----+
  |  1| tom|
  |  2|ammy|
  |  3|mark|
  +---+----+
  ```

– selectExpr

selectExpr is used to perform special processing on a field. For example, the **selectExpr** function can be used to change the field name. The following is an example:

If you want to set the **name** field to **name_test** and add 1 to the value of **age**, run the following statement:

jdbcDF.selectExpr(**"id"**, **"name as name_test"**, **"age+1"**).show()

– col

col is used to obtain a specified field. Different from **select**, **col** can only be used to query the column type and one field can be returned at a time. The following is an example:

val idCol = jdbcDF.col(**"id"**)

– drop

drop is used to delete a specified field. Specify a field you need to delete (only one field can be deleted at a time), the DataFrame object that does not contain the field is returned. Here is an example:

jdbcDF.drop(**"id"**).show()

```
+----+---+
|name|age|
+----+---+
|  qz| 13|
| chm| 13|
| tom| 18|
|ammy| 18|
+----+---+
```

- Submitting a Spark job

a. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

NOTE

- If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.rds** when you submit a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Maven dependency

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- Connecting to data sources through SQL APIs

```
import java.util.Properties
import org.apache.spark.sql.SparkSession

object Test_SQL_RDS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a data table for DLI-associated RDS
    sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (
   'url'='jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306,
      'dbtable'='test.customer',
      'user'='root',
      'password'='######',
          'driver'='com.mysql.jdbc.Driver')")

    //*****************************SQL model***********************************
    //Insert data into the DLI data table
    sparkSession.sql("insert into dli_to_rds values(1,'John',24),(2,'Bob',32)")

    //Read data from DLI data table
    val dataFrame = sparkSession.sql("select * from dli_to_rds")
    dataFrame.show()

    //drop table
    sparkSession.sql("drop table dli_to_rds")

    sparkSession.close()
  }
}
```

- Connecting to data sources through DataFrame APIs

```
import java.util.Properties
import org.apache.spark.sql.SparkSession
```



```
import org.apache.spark.sql.SaveMode

object Test_SQL_RDS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    //*****************************DataFrame model***********************************
    // Set the connection configuration parameters. Contains url, username, password, dbtable.
    val url = "jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306"
    val username = "root"
    val password = "######"
    val dbtable = "test.customer"

    // Create a DataFrame and initialize the DataFrame data.
    var dataFrame_1 = sparkSession.createDataFrame(List((1, "Jack", 18)))

    // Rename the fields set by the createDataFrame() method.
    val df = dataFrame_1.withColumnRenamed("_1", "id")
                  .withColumnRenamed("_2", "name")
                  .withColumnRenamed("_3", "age")

    // Write data to the rds_table_1 table
    df.write.format("jdbc")
      .option("url", url)
      .option("dbtable", dbtable)
      .option("user", username)
      .option("password", password)
      .option("driver", "com.mysql.jdbc.Driver")
      .mode(SaveMode.Append)
      .save()

    // DataFrame object for data manipulation
    //Filter users with id=1
    var newDF = df.filter("id!=1")
    newDF.show()

    // Filter the id column data
    var newDF_1 = df.drop("id")
    newDF_1.show()

    // Read the data of the customer table in the RDS database
    // Way one: Read data from RDS using read.format()
    val jdbcDF = sparkSession.read.format("jdbc")
                  .option("url", url)
                  .option("dbtable", dbtable)
                  .option("user", username)
                  .option("password", password)
                  .option("driver", "com.mysql.jdbc.Driver")
                  .load()
    // Way two: Read data from RDS using read.jdbc()
    val properties = new Properties()
    properties.put("user", username)
    properties.put("password", password)
    val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)

    /**
     * Register the dateFrame read by read.format() or read.jdbc() as a temporary table, and query the
data
     * using the sql statement.
     */
    jdbcDF.registerTempTable("customer_test")
    val result = sparkSession.sql("select * from customer_test where id = 1")
    result.show()

    sparkSession.close()
  }
}
```

- DataFrame-related operations


```
 // The where() method uses " and" and "or" for condition filters, returning filtered DataFrame
objects
 jdbcDF.where("id = 1 or age <=10").show()

 // The filter() method is used in the same way as the where() method.
 jdbcDF.filter("id = 1 or age <=10").show()

 // The select() method passes multiple arguments and returns the DataFrame object of the specified
field.
 jdbcDF.select("id").show()
 jdbcDF.select("id", "name").show()
 jdbcDF.select("id","name").where("id<4").show()

 /**
  * The selectExpr() method implements special handling of fields, such as renaming, increasing or
  * decreasing data values.
  */
 jdbcDF.selectExpr("id", "name as name_test", "age+1").show()

 // The col() method gets a specified field each time, and the return type is a Column type.
 val idCol = jdbcDF.col("id")

 /**
  * The drop() method returns a DataFrame object that does not contain deleted fields, and only one
field
  * can be deleted at a time.
  */
 jdbcDF.drop("id").show()
```

4.3.6.2 PySpark Example Code

Development Description

- Prerequisites

  A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation

  a. Import dependency packages.

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType
  from pyspark.sql import SparkSession
  ```

  b. Create a session.

  ```
  sparkSession = SparkSession.builder.appName("datasource-rds").getOrCreate()
  ```

- Connecting to data sources through DataFrame APIs

  a. Configure datasource connection parameters.

  ```
  url = "jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306"
  dbtable = "test.customer"
  user = "root"
  password = "######"
  driver = "com.mysql.jdbc.Driver"
  ```

  For details about the parameters, see **Table 4-19**.

  b. Set data.

  ```
  dataList = sparkSession.sparkContext.parallelize([(123, "Katie", 19)])
  ```

  c. Configure the schema.


```
schema = StructType([StructField("id", IntegerType(), False),\
                     StructField("name", StringType(), False),\
                     StructField("age", IntegerType(), False)])
```

d. Create a DataFrame.

```
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

e. Save data to RDS.

```
dataFrame.write \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .mode("Append") \
    .save()
```

NOTE

The value of **mode** can be one of the following:

- **ErrorIfExis**: If the data already exists, the system throws an exception.
- **Overwrite**: If the data already exists, the original data will be overwritten.
- **Append**: If the data already exists, the system saves the new data.
- **Ignore**: If the data already exists, no operation is required. This is similar to the SQL statement **CREATE TABLE IF NOT EXISTS**.

f. Read data from RDS.

```
jdbcDF = sparkSession.read \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .load()
jdbcDF.show()
```

g. View the operation result.

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|123|Katie| 19|
+---+-----+---+
```

- Connecting to data sources through SQL APIs

a. Create a table to connect to an RDS data source and set connection parameters.

```
sparkSession.sql(
    "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (\
    'url'='jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306',\
    'dbtable'='test.customer',\
    'user'='root',\
    'password'='######',\
    'driver'='com.mysql.jdbc.Driver')")
```

For details about the parameters for creating a table, see **Table 4-19**.

b. Insert data.

```
sparkSession.sql("insert into dli_to_rds values(3,'John',24)")
```

c. Query data.

```
jdbcDF_after = sparkSession.sql("select * from dli_to_rds")
jdbcDF_after.show()
```

d. View the operation result.


```
+---+-----+---+
| id| name|age|
+---+-----+---+
|123|Katie| 19|
|  3| John| 24|
+---+-----+---+
```

- Submitting a Spark job
  a. Upload the Python code file to the OBS bucket.
  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.
  c. After the Spark job is created, click **Execute** in the upper right corner of the console to submit the job. If the message "Spark job submitted successfully." is displayed, the Spark job is successfully submitted. You can view the status and logs of the submitted job on the **Spark Jobs** page.

  NOTE

  - The queue you select for creating a Spark job is the one bound when you create the datasource connection.
  - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.rds** when you submit a job.
  - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

NOTE

If the following sample code is directly copied to the **.py** file, note that unexpected characters may exist after the backslashes (\) in the file content. You need to delete the indentations or spaces after the backslashes (\).

- Connecting to data sources through DataFrame APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession
if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-rds").getOrCreate()

  # Set cross-source connection parameters.
  url = "jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306"
  dbtable = "test.customer"
  user = "root"
  password = "######"
  driver = "com.mysql.jdbc.Driver"

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([(123, "Katie", 19)])
```



```
# Setting schema
schema = StructType([StructField("id", IntegerType(), False),\
                     StructField("name", StringType(), False),\
                     StructField("age", IntegerType(), False)])

# Create a DataFrame from RDD and schema
dataFrame = sparkSession.createDataFrame(dataList, schema)

# Write data to the RDS.
dataFrame.write \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .mode("Append") \
    .save()

# Read data
jdbcDF = sparkSession.read \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .load()
jdbcDF.show()

# close session
sparkSession.stop()
```

- Connecting to data sources through SQL APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-rds").getOrCreate()

  # Create a data table for DLI - associated RDS
  sparkSession.sql(
       "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (\
       'url'='jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306',\
       'dbtable'='test.customer',\
       'user'='root',\
       'password'='######',\
       'driver'='com.mysql.jdbc.Driver')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into dli_to_rds values(3,'John',24)")

  # Read data from DLI data table
  jdbcDF = sparkSession.sql("select * from dli_to_rds")
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

4.3.6.3 Java Example Code

Development Description

- Prerequisites

A datasource connection has been created and bound to a queue on the DLI management console. For details, see **Enhanced Datasource Connections**.

> **NOTE**
>
> Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation

  a. Import dependencies.

  - Maven dependency involved

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

  - Import dependency packages.

    ```
    import org.apache.spark.sql.SparkSession;
    ```

  b. Create a session.

  ```
  SparkSession sparkSession = SparkSession.builder().appName("datasource-rds").getOrCreate();
  ```

- Connecting to data sources through SQL APIs

  – Create a table to connect to an RDS data source and set connection parameters.

  ```
  sparkSession.sql(
    "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (
       'url'='jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306',  // Set this parameter to the actual URL.
       'dbtable'='test.customer',
       'user'='root',  // Set this parameter to the actual user.
       'password'='######',  // Set this parameter to the actual password.
       'driver'='com.mysql.jdbc.Driver')")
  ```

  For details about the parameters for creating a table, see **Table 4-19**.

  – Insert data.

  ```
  sparkSession.sql("insert into dli_to_rds values (1,'John',24)");
  ```

  – Query data.

  ```
  sparkSession.sql("select * from dli_to_rd").show();
  ```

  Response

  ```
  +---+----+---+
  | id|name|age|
  +---+----+---+
  |  1|John| 24|
  +---+----+---+
  ```

- Submitting a Spark job

  a. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  c. After the Spark job is created, click **Execute** in the upper right corner of the console to submit the job. If the message "Spark job submitted successfully." is displayed, the Spark job is successfully submitted. You can view the status and logs of the submitted job on the **Spark Jobs** page.

NOTE

- If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.rds** when you submit a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.SparkSession;

public class java_rds {

    public static void main(String[] args) {
        SparkSession sparkSession = SparkSession.builder().appName("datasource-rds").getOrCreate();

        // Create a data table for DLI-associated RDS
        sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS ('url'='jdbc:mysql://
192.168.6.150:3306','dbtable'='test.customer','user'='root','password'='**','driver'='com.mysql.jdbc.Driver')");

        //*****************************SQL model***********************************
        //Insert data into the DLI data table
        sparkSession.sql("insert into dli_to_rds values(3,'Liu',21),(4,'Joey',34)");

        //Read data from DLI data table
        sparkSession.sql("select * from dli_to_rds");

        //drop table
        sparkSession.sql("drop table dli_to_rds");

        sparkSession.close();
    }
}
```

4.3.7 Connecting to Redis

4.3.7.1 Scala Example Code

Development Description

Redis supports only enhanced datasource connections.

- Prerequisites

  An enhanced datasource connection has been created on the DLI management console and bound to a queue in packages. For details, see **Enhanced Datasource Connections**.

> **NOTE**
>
> Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Constructing dependency information and creating a Spark session

  a. Import dependencies.

  Maven dependency involved

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.1.0</version>
</dependency>
<dependency>
  <groupId>com.redislabs</groupId>
  <artifactId>spark-redis</artifactId>
  <version>2.4.0</version>
</dependency>
```

  Import dependency packages.

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types._
import com.redislabs.provider.redis._
import scala.reflect.runtime.universe._
import org.apache.spark.{SparkConf, SparkContext}
```

- Connecting to data sources through DataFrame APIs

  a. Create a session.

```
val sparkSession = SparkSession.builder().appName("datasource_redis").getOrCreate()
```

  b. Construct a schema.

```
//method one
var schema = StructType(Seq(StructField("name", StringType, false), StructField("age",
IntegerType, false)))
var rdd = sparkSession.sparkContext.parallelize(Seq(Row("abc",34),Row("Bob",19)))
var dataFrame = sparkSession.createDataFrame(rdd, schema)
// //method two
// var jdbcDF= sparkSession.createDataFrame(Seq(("Jack",23)))
// val dataFrame = jdbcDF.withColumnRenamed("_1", "name").withColumnRenamed("_2",
"age")
// //method three
// case class Person(name: String, age: Int)
// val dataFrame = sparkSession.createDataFrame(Seq(Person("John", 30), Person("Peter", 45)))
```

  > **NOTE**
  >
  > **case class Person(name: String, age: Int)** must be written outside the object. For details, see **Connecting to data sources through DataFrame APIs**.

  c. Import data to Redis.

```
dataFrame .write
  .format("redis")
  .option("host","192.168.4.199")
  .option("port","6379")
  .option("table","person")
  .option("password","******")
  .option("key.column","name")
  .mode(SaveMode.Overwrite)
  .save()
```

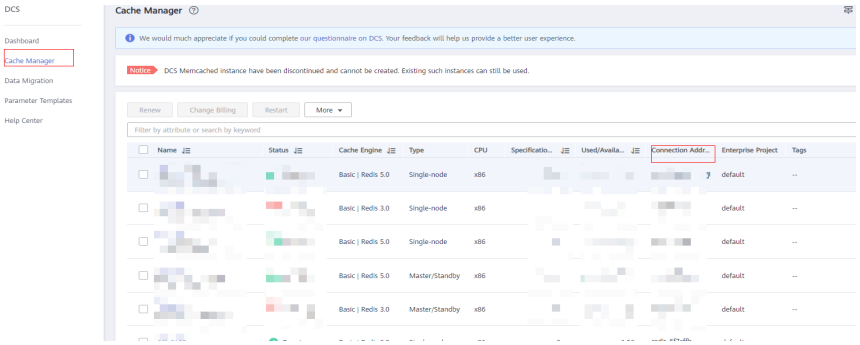

Table 4-20 Redis operation parameters

Parameter	Description
host	IP address of the Redis cluster to be connected. To obtain the IP address, log in to the Huawei Cloud official website, search for **redis**, go to the console of Distributed Cache Service for Redis, and choose **Cache Manager**. Select an IP address (including the port information) based on the IP address required by the host name to copy the data. For details, see **Figure 4-41**.
port	Access port.
password	Password for the connection. This parameter is optional if no password is required.
table	Key or hash key in Redis. • This parameter is mandatory when Redis data is inserted. • Either this parameter or the **keys.pattern** parameter when Redis data is queried.
keys.pattern	Use a regular expression to match multiple keys or hash keys. This parameter is used only for query. Either this parameter or **table** is used to query Redis data.
key.column	Key value of a column. This parameter is optional. If a key is specified when data is written, the key must be specified during query. Otherwise, the key will be abnormally loaded during query.
partitions.number	Number of concurrent tasks during data reading.
scan.count	Number of data records read in each batch. The default value is **100**. If the CPU usage of the Redis cluster still needs to be improved during data reading, increase the value of this parameter.
iterator.grouping.size	Number of data records inserted in each batch. The default value is **100**. If the CPU usage of the Redis cluster still needs to be improved during the insertion, increase the value of this parameter.
timeout	Timeout interval for connecting to the Redis, in milliseconds. The default value is **2000** (2 seconds).

NOTE

- The options of **mode** are **Overwrite**, **Append**, **ErrorIfExis**, and **Ignore**.
- To save nested DataFrames, use **.option("model", "binary")**.
- Specify the data expiration time by **.option("ttl", 1000)**. The unit is second.

Figure 4-41 Obtaining the IP address and port number of Redis



d. Read data from Redis.

```
sparkSession.read
  .format("redis")
  .option("host","192.168.4.199")
  .option("port","6379")
  .option("table", "person")
  .option("password","######")
  .option("key.column","name")
  .load()
  .show()
```

Operation result

```
+------+---+\n
|  name|age|\n
+------+---+\n
|huawei| 34|\n
+------+---+\n
|   Bob| 19|\n
+------+---+\n\n
```

- Connecting to data sources using Spark RDDs

a. Create a datasource connection.

```
val sparkContext = new SparkContext(new SparkConf()
  .setAppName("datasource_redis")
  .set("spark.redis.host", "192.168.4.199")
  .set("spark.redis.port", "6379")
  .set("spark.redis.auth", "######")
  .set("spark.driver.allowMultipleContexts","true"))
```

NOTE

If **spark.driver.allowMultipleContexts** is set to **true**, only the current context is used when multiple contexts are started, to prevent context invoking conflicts.

b. Insert data.

i. Save data in strings.

```
val stringRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("high","111"), ("together","333")))
sparkContext.toRedisKV(stringRedisData)
```

ii. Save data in hashes.


```
val hashRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("saprk","123"), ("data","222")))
sparkContext.toRedisHASH(hashRedisData, "hashRDD")
```

iii. Save data in lists.

```
val data = List(("school","112"), ("tom","333"))
val listRedisData:RDD[String] = sparkContext.parallelize(Seq[(String)](data.toString()))
sparkContext.toRedisLIST(listRedisData, "listRDD")
```

iv. Save data in sets.

```
val setData = Set(("bob","133"),("kity","322"))
val setRedisData:RDD[(String)] = sparkContext.parallelize(Seq[(String)]
(setData.mkString))
sparkContext.toRedisSET(setRedisData, "setRDD")
```

v. Save data in zsets.

```
val zsetRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("whight","234"), ("bobo","343")))
sparkContext.toRedisZSET(zsetRedisData, "zsetRDD")
```

c. Query data.

i. Query data by traversing keys.

```
val keysRDD = sparkContext.fromRedisKeys(Array("high","together", "hashRDD",
"listRDD", "setRDD","zsetRDD"), 6)
keysRDD.getKV().collect().foreach(println)
keysRDD.getHash().collect().foreach(println)
keysRDD.getList().collect().foreach(println)
keysRDD.getSet().collect().foreach(println)
keysRDD.getZSet().collect().foreach(println)
```

ii. Query data by string.

```
sparkContext.fromRedisKV(Array( "high","together")).collect().foreach{println}
```

iii. Query data by hash.

```
sparkContext.fromRedisHash(Array("hashRDD")).collect().foreach{println}
```

iv. Query data by list.

```
sparkContext.fromRedisList(Array("listRDD")).collect().foreach{println}
```

v. Query data by set.

```
sparkContext.fromRedisSet(Array("setRDD")).collect().foreach{println}
```

vi. Query data by zset.

```
sparkContext.fromRedisZSet(Array("zsetRDD")).collect().foreach{println}
```

- Connecting to data sources through SQL APIs

a. Create a table to connect to a Redis data source.

```
sparkSession.sql(
    "CREATE TEMPORARY VIEW person (name STRING, age INT) USING
org.apache.spark.sql.redis OPTIONS (
    'host' = '192.168.4.199',
    'port' = '6379',
    'password' = '######',
    table  'person')".stripMargin)
```

b. Insert data.

```
sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)
```

c. Query data.

```
sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)
```

- Submitting a Spark job

a. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

For details about console operations, see **Creating a Spark Job**. For details about API operations, see **Creating a Batch Processing Job**.

NOTE

- If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.redis** when you submit a job.
- If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*
- For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
- For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Maven dependency

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.1.0</version>
</dependency>
<dependency>
  <groupId>com.redislabs</groupId>
  <artifactId>spark-redis</artifactId>
  <version>2.4.0</version>
</dependency>
```

- Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.{SparkSession};

object Test_Redis_SQL {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().appName("datasource_redis").getOrCreate();

    sparkSession.sql(
      "CREATE TEMPORARY VIEW person (name STRING, age INT) USING org.apache.spark.sql.redis
OPTIONS (
        'host' = '192.168.4.199', 'port' = '6379', 'password' = '******',table 'person')".stripMargin)

    sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)

    sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)

    sparkSession.close()
  }
}
```

- Connecting to data sources through DataFrame APIs

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types._

object Test_Redis_SparkSql {
  def main(args: Array[String]): Unit = {
  // Create a SparkSession session.
  val sparkSession = SparkSession.builder().appName("datasource_redis").getOrCreate()
```



```
  // Set cross-source connection parameters.
  val host = "192.168.4.199"
  val port = "6379"
  val table = "person"
  val auth = "######"
  val key_column = "name"

  // ******** setting DataFrame ********
  // method one
  var schema = StructType(Seq(StructField("name", StringType, false),StructField("age", IntegerType,
false)))
  var rdd = sparkSession.sparkContext.parallelize(Seq(Row("huawei",34),Row("Bob",19)))
  var dataFrame = sparkSession.createDataFrame(rdd, schema)

// // method two
// var jdbcDF= sparkSession.createDataFrame(Seq(("Jack",23)))
// val dataFrame = jdbcDF.withColumnRenamed("_1", "name").withColumnRenamed("_2", "age")

// // method three
// val dataFrame = sparkSession.createDataFrame(Seq(Person("John", 30), Person("Peter", 45)))

  // Write data to redis
  dataFrame.write.format("redis").option("host",host).option("port",port).option("table",
table).option("password",auth).mode(SaveMode.Overwrite).save()

  // Read data from redis
  sparkSession.read.format("redis").option("host",host).option("port",port).option("table",
table).option("password",auth).load().show()

  // Close session
  sparkSession.close()
  }
}
// methoe two
// case class Person(name: String, age: Int)
```

- Connecting to data sources using Spark RDDs

```
import com.redislabs.provider.redis._
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

object Test_Redis_RDD {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkContext = new SparkContext(new SparkConf()
          .setAppName("datasource_redis")
          .set("spark.redis.host", "192.168.4.199")
          .set("spark.redis.port", "6379")
          .set("spark.redis.auth", "@@@@@@")
          .set("spark.driver.allowMultipleContexts","true"))

    //**************** Write data to redis **********************
    // Save String type data
    val stringRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("high","111"), ("together","333")))
    sparkContext.toRedisKV(stringRedisData)

    // Save Hash type data
    val hashRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("saprk","123"), ("data","222")))
    sparkContext.toRedisHASH(hashRedisData, "hashRDD")

    // Save List type data
    val data = List(("school","112"), ("tom","333"));
    val listRedisData:RDD[String] = sparkContext.parallelize(Seq[(String)](data.toString()))
    sparkContext.toRedisLIST(listRedisData, "listRDD")

    // Save Set type data
    val setData = Set(("bob","133"),("kity","322"))
    val setRedisData:RDD[(String)] = sparkContext.parallelize(Seq[(String)](setData.mkString))
```



```
    sparkContext.toRedisSET(setRedisData, "setRDD")

    // Save ZSet type data
    val zsetRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("whight","234"), ("bobo","343")))
    sparkContext.toRedisZSET(zsetRedisData, "zsetRDD")

    // ***************************** Read data from redis *****************************************
    // Traverse the specified key and get the value
    val keysRDD = sparkContext.fromRedisKeys(Array("high","together", "hashRDD", "listRDD",
"setRDD","zsetRDD"), 6)
    keysRDD.getKV().collect().foreach(println)
    keysRDD.getHash().collect().foreach(println)
    keysRDD.getList().collect().foreach(println)
    keysRDD.getSet().collect().foreach(println)
    keysRDD.getZSet().collect().foreach(println)

    // Read String type data//
    val stringRDD = sparkContext.fromRedisKV("keyPattern *")
    sparkContext.fromRedisKV(Array( "high","together")).collect().foreach{println}

    // Read Hash type data//
    val hashRDD = sparkContext.fromRedisHash("keyPattern *")
    sparkContext.fromRedisHash(Array("hashRDD")).collect().foreach{println}

    // Read List type data//
    val listRDD = sparkContext.fromRedisList("keyPattern *")
    sparkContext.fromRedisList(Array("listRDD")).collect().foreach{println}

    // Read Set type data//
    val setRDD = sparkContext.fromRedisSet("keyPattern *")
    sparkContext.fromRedisSet(Array("setRDD")).collect().foreach{println}

    // Read ZSet type data//
    val zsetRDD = sparkContext.fromRedisZSet("keyPattern *")
    sparkContext.fromRedisZSet(Array("zsetRDD")).collect().foreach{println}

    // close session
    sparkContext.stop()
  }
}
```

4.3.7.2 PySpark Example Code

Development Description

Redis supports only enhanced datasource connections.

- Prerequisites

  An enhanced datasource connection has been created on the DLI management console and bound to a queue in packages. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Connecting to data sources through DataFrame APIs

  a. Import dependencies.

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType
  from pyspark.sql import SparkSession
  ```


b. Create a session.

```
sparkSession = SparkSession.builder.appName("datasource-redis").getOrCreate()
```

c. Set connection parameters.

```
host = "192.168.4.199"
port = "6379"
table = "person"
auth = "@@@@@@"
```

d. Create a DataFrame.

i. Method 1:

```
dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19),(2,"Tom",20)])
schema = StructType([StructField("id", IntegerType(), False),
                     StructField("name", StringType(), False),
                     StructField("age", IntegerType(), False)])
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

ii. Method 2:

```
jdbcDF = sparkSession.createDataFrame([(3,"Jack", 23)])
dataFrame = jdbcDF.withColumnRenamed("_1", "id").withColumnRenamed("_2",
"name").withColumnRenamed("_3", "age")
```

e. Import data to Redis.

```
dataFrame.write
 .format("redis")\
 .option("host", host)\
 .option("port", port)\
 .option("table", table)\
 .option("password", auth)\
 .mode("Overwrite")\
 .save()
```

NOTE

- The options of **mode** are **Overwrite**, **Append**, **ErrorIfExis**, and **Ignore**.
- To specify a key, use **.option("key.column", "name")**. **name** indicates the column name.
- To save nested DataFrames, use **.option("model", "binary")**.
- If you need to specify the data expiration time, use **.option("ttl", 1000)**. The unit is second.

f. Read data from Redis.

```
sparkSession.read.format("redis").option("host", host).option("port", port).option("table",
table).option("password", auth).load().show()
```

g. View the operation result.

```
+---+-----+---+\n
| id| name|age|\n
+---+-----+---+\n
|  2|  Tom| 20|\n
|  1|Katie| 19|\n
+---+-----+---+\n\n
```

- Connecting to data sources through SQL APIs

a. Create a table to connect to a Redis data source.

```
sparkSession.sql(
    "CREATE TEMPORARY VIEW person (name STRING, age INT) USING
org.apache.spark.sql.redis OPTIONS (
    'host' = '192.168.4.199',
    'port' = '6379',
    'password' = '######',
    table  'person')".stripMargin)
```

b. Insert data.

```
sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)
```


c. Query data.

```
sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)
```

- Submitting a Spark job

  a. Upload the Python code file to the OBS bucket.

  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  NOTE

  - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.redis** when you submit a job.
  - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Connecting to data sources through DataFrame APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession
if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-redis").getOrCreate()

  # Set cross-source connection parameters.
  host = "192.168.4.199"
  port = "6379"
  table = "person"
  auth = "######"

  # Create a DataFrame and initialize the DataFrame data.
  # *******   method noe   *********
  dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19),(2,"Tom",20)])
  schema = StructType([StructField("id", IntegerType(), False),StructField("name", StringType(), False),StructField("age", IntegerType(), False)])
  dataFrame_one = sparkSession.createDataFrame(dataList, schema)

  # ****** method two ******
  # jdbcDF = sparkSession.createDataFrame([(3,"Jack", 23)])
  # dataFrame = jdbcDF.withColumnRenamed("_1", "id").withColumnRenamed("_2", "name").withColumnRenamed("_3", "age")

  # Write data to the redis table
  dataFrame.write.format("redis").option("host", host).option("port", port).option("table", table).option("password", auth).mode("Overwrite").save()
  # Read data
  sparkSession.read.format("redis").option("host", host).option("port", port).option("table", table).option("password", auth).load().show()

  # close session
  sparkSession.stop()
```


- Connecting to data sources through SQL APIs

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession
  sparkSession = SparkSession.builder.appName("datasource_redis").getOrCreate()

  sparkSession.sql(
    "CREATE TEMPORARY VIEW person (name STRING, age INT) USING org.apache.spark.sql.redis
OPTIONS (\
    'host' = '192.168.4.199', \
    'port' = '6379',\
    'password' = '######',\
    'table'= 'person')".stripMargin);

  sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)

  sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)

  # close session
  sparkSession.stop()
```

4.3.7.3 Java Example Code

Development Description

Redis supports only enhanced datasource connections.

- Prerequisites

  An enhanced datasource connection has been created on the DLI management console and bound to a queue in packages. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation

  a. Import dependencies.

  - Maven dependency involved

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  - Import dependency packages.

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.sql.*;
import org.apache.spark.sql.types.DataTypes;
import org.apache.spark.sql.types.StructField;
import org.apache.spark.sql.types.StructType;
import java.util.*;
```

  b. Create a session.

```
SparkConf sparkConf = new SparkConf();
sparkConf.setAppName("datasource-redis")
```



```
        .set("spark.redis.host", "192.168.4.199")
        .set("spark.redis.port", "6379")
        .set("spark.redis.auth", "******")
        .set("spark.driver.allowMultipleContexts","true");
JavaSparkContext javaSparkContext = new JavaSparkContext(sparkConf);
SQLContext sqlContext = new SQLContext(javaSparkContext);
```

- Connecting to data sources through DataFrame APIs

  a. Read JSON data as DataFrames.

  ```
  JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList(
        "{\"id\":\"1\",\"name\":\"Ann\",\"age\":\"18\"}",
        "{\"id\":\"2\",\"name\":\"lisi\",\"age\":\"21\"}"));
  Dataset dataFrame = sqlContext.read().json(javaRDD);
  ```

  b. Construct the Redis connection parameters.

  ```
  Map map = new HashMap<String, String>();
  map.put("table","person");
  map.put("key.column","id");
  ```

  c. Save data to Redis.

  ```
  dataFrame.write().format("redis").options(map).mode(SaveMode.Overwrite).save();
  ```

  d. Read data from Redis.

  ```
  sqlContext.read().format("redis").options(map).load().show();
  ```

  e. View the operation result.

  ```
  +---+--------+---+\n
  | id|    name|age|\n
  +---+--------+---+\n
  |  1|zhangsan| 18|\n
  +---+--------+---+\n
  |  2|    lisi| 21|\n
  +---+--------+---+\n\n
  ```

- Submitting a Spark job

  a. Upload the Java code file to the OBS bucket.

  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  > NOTE
  >
  > - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.redis** when you submit a job.
  > - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.
  >
  >   spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*
  >
  >   spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*
  > - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  > - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

```
public class Test_Redis_DaraFrame {
  public static void main(String[] args) {
    //create a SparkSession session
    SparkConf sparkConf = new SparkConf();
```



```
  sparkConf.setAppName("datasource-redis")
          .set("spark.redis.host", "192.168.4.199")
          .set("spark.redis.port", "6379")
          .set("spark.redis.auth", "*******")
          .set("spark.driver.allowMultipleContexts","true");
  JavaSparkContext javaSparkContext = new JavaSparkContext(sparkConf);
  SQLContext sqlContext = new SQLContext(javaSparkContext);

  //Read RDD in JSON format to create DataFrame
  JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList(
          "{\"id\":\"1\",\"name\":\"Ann\",\"age\":\"18\"}",
          "{\"id\":\"2\",\"name\":\"lisi\",\"age\":\"21\"}"));
  Dataset dataFrame = sqlContext.read().json(javaRDD);

  Map map = new HashMap<String, String>();
  map.put("table","person");
  map.put("key.column","id");
  dataFrame.write().format("redis").options(map).mode(SaveMode.Overwrite).save();
  sqlContext.read().format("redis").options(map).load().show();

 }
}
```

4.3.7.4 Troubleshooting

Problem 1

- Symptom

  After the code is directly copied to the **.py** file, unexpected characters may exist after the backslashes (\).

- Solution

  Delete the indentations or spaces after the backslashes (\).

4.3.8 Connecting to Mongo

4.3.8.1 Scala Example Code

Development Description

Mongo can be connected only through enhanced datasource connections.

NOTE

DDS is compatible with the MongoDB protocol.

An enhanced datasource connection has been created on the DLI management console and bound to a queue in packages. For details, see **Enhanced Datasource Connections**.

NOTE

Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Constructing dependency information and creating a Spark session
  a. Import dependencies.

     Maven dependency involved


```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

Import dependency packages.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}
```

Create a session.

```
val sparkSession = SparkSession.builder().appName("datasource-mongo").getOrCreate()
```

- Connecting to data sources through SQL APIs

  a. Create a table to connect to a Mongo data source.

```
sparkSession.sql(
  "create table test_dds(id string, name string, age int) using mongo options(
    'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',
    'uri' = 'mongodb://username:pwd@host:8635/db',
    'database' = 'test',
    'collection' = 'test',
    'user' = 'rwuser',
    'password' = '######')")
```

Table 4-21 Parameters for creating a table

Parameter	Description
url	• URL format: "IP:PORT[,IP:PORT]/[DATABASE][.COLLECTION][AUTH_PROPERTIES]" Example: "192.168.4.62:8635/test?authSource=admin" • The URL needs to be obtained from the Mongo (DDS) connection address.. The obtained Mongo connection address is in the following format: ***Protocol header*://*Username*:*Password*@*Connection address*:*Port number*/*Database name*?authSource=admin** Example: mongodb://rwuser:****@192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin
uri	URI format: **mongodb://username:pwd@host:8635/db** Set the following parameters to the actual values: • **username**: username used for creating the Mongo (DDS) database • **pwd**: password of the username for the Mongo (DDS) database • **host**: IP address of the Mongo (DDS) database instance • **db**: name of the created Mongo (DDS) database For details about how to create a Mongo (DDS) database user, see **Creating a Database Account Using Commands**.
database	DDS database name. If the database name is specified in the URL, the database name in the URL does not take effect.

Parameter	Description
collection	Collection name in the DDS. If the collection is specified in the URL, the collection in the URL does not take effect. **NOTE** If a collection already exists in DDS, you do not need to specify schema information when creating a table. DLI automatically generates schema information based on data in the collection.
user	Username for accessing the DDS cluster.
password	Password for accessing the DDS cluster.

b. Insert data.

```
sparkSession.sql("insert into test_dds values('3', 'Ann',23)")
```

c. Query data.

```
sparkSession.sql("select * from test_dds").show()
```

View the operation result.

```
+---+--------+---+\n
| id|    name|age|\n
+---+--------+---+\n
|  2|     Bob| 32|\n
|  1|    John| 23|\n
|  3|zhangsan| 23|\n
+---+--------+---+\n\n
```

- Connecting to data sources through DataFrame APIs

a. Set connection parameters.

```
val url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
val uri = "mongodb://username:pwd@host:8635/db"
val user = "rwuser"
val database = "test"
val collection = "test"
val password = "######"
```

b. Construct a schema.

```
val schema = StructType(List(StructField("id", StringType), StructField("name", StringType), StructField("age", IntegerType)))
```

c. Construct a DataFrame.

```
val rdd = spark.sparkContext.parallelize(Seq(Row("1", "John", 23), Row("2", "Bob", 32)))
val dataFrame = spark.createDataFrame(rdd, schema)
```

d. Import data to Mongo.

```
dataFrame.write.format("mongo")
  .option("url", url)
  .option("uri", uri)
  .option("database", database)
  .option("collection", collection)
  .option("user", user)
  .option("password", password)
  .mode(SaveMode.Overwrite)
  .save()
```

NOTE

The options of **mode** are **Overwrite**, **Append**, **ErrorIfExis**, and **Ignore**.

e. Read data from Mongo.

```
val jdbcDF = spark.read.format("mongo").schema(schema)
  .option("url", url)
  .option("uri", uri)
  .option("database", database)
  .option("collection", collection)
  .option("user", user)
  .option("password", password)
  .load()
```

Operation result

```
+---+----+---+\n
| id|name|age|\n
+---+----+---+\n
|  2| Bob| 32|\n
|  1|John| 23|\n
+---+----+---+\n\n
```

- Submitting a Spark job

  a. Generate a JAR file based on the code file and upload the JAR file to the OBS bucket.

  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  For details about console operations, see **Creating a Spark Job**. For details about API operations, see **Creating a Batch Processing Job**.

  NOTE

  - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.mongo** when you submit a job.
  - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Maven dependency

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- Connecting to data sources through SQL APIs

```
import org.apache.spark.sql.SparkSession

object TestMongoSql {
  def main(args: Array[String]): Unit = {
    val sparkSession = SparkSession.builder().getOrCreate()
    sparkSession.sql(
      "create table test_dds(id string, name string, age int) using mongo options(
```



```
      'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',
      'uri' = 'mongodb://username:pwd@host:8635/db',
      'database' = 'test',
      'collection' = 'test',
      'user' = 'rwuser',
      'password' = '######')")
   sparkSession.sql("insert into test_dds values('3', 'Ann',23)")
   sparkSession.sql("select * from test_dds").show()
   sparkSession.close()
  }
}
```

- Connecting to data sources through DataFrame APIs

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}

object Test_Mongo_SparkSql {
  def main(args: Array[String]): Unit = {
  //  Create a SparkSession session.
  val spark = SparkSession.builder().appName("mongodbTest").getOrCreate()

  // Set the connection configuration parameters.
  val url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
  val uri = "mongodb://username:pwd@host:8635/db"
  val user = "rwuser"
  val database = "test"
  val collection = "test"
  val password = "######"

  // Setting up the schema
  val schema = StructType(List(StructField("id", StringType), StructField("name", StringType),
StructField("age", IntegerType)))

  // Setting up the DataFrame
  val rdd = spark.sparkContext.parallelize(Seq(Row("1", "John", 23), Row("2", "Bob", 32)))
  val dataFrame = spark.createDataFrame(rdd, schema)


  // Write data to mongo
  dataFrame.write.format("mongo")
    .option("url", url)
    .option("uri", uri)
    .option("database", database)
    .option("collection", collection)
    .option("user", user)
    .option("password", password)
    .mode(SaveMode.Overwrite)
    .save()

  // Reading data from mongo
  val jdbcDF = spark.read.format("mongo").schema(schema)
    .option("url", url)
    .option("uri", uri)
    .option("database", database)
    .option("collection", collection)
    .option("user", user)
    .option("password", password)
    .load()
  jdbcDF.show()

  spark.close()
 }
}
```


4.3.8.2 PySpark Example Code

Development Description

Mongo can be connected only through enhanced datasource connections.

NOTE

DDS is compatible with the MongoDB protocol.

- Prerequisites

  An enhanced datasource connection has been created on the DLI management console and bound to a queue in packages. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Connecting to data sources through DataFrame APIs

  a. Import dependencies.

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType
  from pyspark.sql import SparkSession
  ```

  b. Create a session.

  ```
  sparkSession = SparkSession.builder.appName("datasource-mongo").getOrCreate()
  ```

  c. Set connection parameters.

  ```
  url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
  uri = "mongodb://username:pwd@host:8635/db"
  user = "rwuser"
  database = "test"
  collection = "test"
  password = "######"
  ```

  NOTE

  For details about the parameters, see **Table 4-21**.

  d. Create a DataFrame.

  ```
  dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19),(2,"Tom",20)])
  schema = StructType([StructField("id", IntegerType(), False),
                       StructField("name", StringType(), False),
                       StructField("age", IntegerType(), False)])
  dataFrame = sparkSession.createDataFrame(dataList, schema)
  ```

  e. Import data to Mongo.

  ```
  dataFrame.write.format("mongo")
    .option("url", url)\
    .option("uri", uri)\
    .option("user",user)\
    .option("password",password)\
    .option("database",database)\
    .option("collection",collection)\
    .mode("Overwrite")\
    .save()
  ```

  f. Read data from Mongo.

  ```
  jdbcDF = sparkSession.read
    .format("mongo")\
    .option("url", url)\
    .option("uri", uri)\
    .option("user",user)\
  ```



```
 .option("password",password)\
 .option("database",database)\
 .option("collection",collection)\
 .load()
jdbcDF.show()
```

g. View the operation result.

```
+---+-----+---+\n
| id| name|age|\n
+---+-----+---+\n
|  2|  Tom| 20|\n
|  1|Katie| 19|\n
+---+-----+---+\n\n
```

- Connecting to data sources through SQL APIs

  a. Create a table to connect to a Mongo data source.

  ```
  sparkSession.sql(
      "create table test_dds(id string, name string, age int) using mongo options(
      'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',
      'uri' = 'mongodb://username:pwd@host:8635/db',
      'database' = 'test',
      'collection' = 'test',
      'user' = 'rwuser',
      'password' = '######')")
  ```

  NOTE

  For details about the parameters, see **Table 4-21**.

  b. Insert data.

  ```
  sparkSession.sql("insert into test_dds values('3', 'Ann',23)")
  ```

  c. Query data.

  ```
  sparkSession.sql("select * from test_dds").show()
  ```

- Submitting a Spark job

  a. Upload the Python code file to the OBS bucket.

  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  NOTE

  - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.mongo** when you submit a job.
  - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

- Connecting to data sources through DataFrame APIs

```
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
```



```
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-mongo").getOrCreate()

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([("1", "Katie", 19),("2","Tom",20)])

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),StructField("name", StringType(), False),
StructField("age", IntegerType(), False)])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(dataList, schema)

  # Setting connection parameters
  url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
  uri = "mongodb://username:pwd@host:8635/db"
  user = "rwuser"
  database = "test"
  collection = "test"
  password = "######"

  # Write data to the mongodb table
  dataFrame.write.format("mongo")
    .option("url", url)\
    .option("uri", uri)\
    .option("user",user)\
    .option("password",password)\
    .option("database",database)\
    .option("collection",collection)
    .mode("Overwrite").save()

  # Read data
  jdbcDF = sparkSession.read.format("mongo")
    .option("url", url)\
    .option("uri", uri)\
    .option("user",user)\
    .option("password",password)\
    .option("database",database)\
    .option("collection",collection)\
    .load()
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

- Connecting to data sources through SQL APIs

```
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-mongo").getOrCreate()

  # Create a data table for DLI - associated mongo
    sparkSession.sql(
      "create table test_dds(id string, name string, age int) using mongo options(\
      'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',\
      'uri' = 'mongodb://username:pwd@host:8635/db',\
      'database' = 'test',\
      'collection' = 'test', \
      'user' = 'rwuser', \
      'password' = '######')")

  # Insert data into the DLI-table
  sparkSession.sql("insert into test_dds values('3', 'Ann',23)")

  # Read data from DLI-table
```



```
sparkSession.sql("select * from test_dds").show()

# close session
sparkSession.stop()
```

4.3.8.3 Java Example Code

Development Description

Mongo can be connected only through enhanced datasource connections.

NOTE

DDS is compatible with the MongoDB protocol.

- Prerequisites

  An enhanced datasource connection has been created on the DLI management console and bound to a queue in packages. For details, see **Enhanced Datasource Connections**.

  NOTE

  Hard-coded or plaintext passwords pose significant security risks. To ensure security, encrypt your passwords, store them in configuration files or environment variables, and decrypt them when needed.

- Code implementation

  a. Import dependencies.

  - Maven dependency involved

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

  - Import dependency packages.

    ```
    import org.apache.spark.SparkConf;
    import org.apache.spark.SparkContext;
    import org.apache.spark.api.java.JavaRDD;
    import org.apache.spark.api.java.JavaSparkContext;
    import org.apache.spark.sql.Dataset;
    import org.apache.spark.sql.Row;
    import org.apache.spark.sql.SQLContext;
    import org.apache.spark.sql.SaveMode;
    ```

  b. Create a session.

  ```
  SparkContext sparkContext = new SparkContext(new SparkConf().setAppName("datasource-mongo"));
  JavaSparkContext javaSparkContext = new JavaSparkContext(sparkContext);
  SQLContext sqlContext = new SQLContext(javaSparkContext);
  ```

- Connecting to data sources through DataFrame APIs

  a. Read JSON data as DataFrames.

  ```
  JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList("{\"id\":\"5\",\"name\":\"Ann\",\"age\":\"23\"}"));
  Dataset<Row> dataFrame = sqlContext.read().json(javaRDD);
  ```

  b. Set connection parameters.

  ```
  String url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin";
  String uri = "mongodb://username:pwd@host:8635/db";
  String user = "rwuser";
  String database = "test";
  ```



```
String collection = "test";
String password = "######";
```

NOTE

For details about the parameters, see **Table 4-21**.

c. Import data to Mongo.

```
dataFrame.write().format("mongo")
    .option("url",url)
    .option("uri",uri)
    .option("database",database)
    .option("collection",collection)
    .option("user",user)
    .option("password",password)
    .mode(SaveMode.Overwrite)
    .save();
```

d. Read data from Mongo.

```
sqlContext.read().format("mongo")
   .option("url",url)
   .option("uri",uri)
   .option("database",database)
   .option("collection",collection)
   .option("user",user)
   .option("password",password)
   .load().show();
```

e. View the operation result.

```
+---+--------+---+\n
| id|    name|age|\n
+---+--------+---+\n
|  5|zhangsan| 23|\n
+---+--------+---+\n\n
```

- Submitting a Spark job

  a. Upload the Java code file to the OBS bucket.

  b. In the Spark job editor, select the corresponding dependency module and execute the Spark job.

  NOTE

  - If the Spark version is 2.3.2 (will be offline soon) or 2.4.5, specify the **Module** to **sys.datasource.mongo** when you submit a job.
  - If the Spark version is 3.1.1 or later, you do not need to select a module. Configure **Spark parameters (--conf)**.

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*
  - For how to submit a job on the console, see Table 3 "Parameters for selecting dependency resources" in **Creating a Spark Job**.
  - For details about how to submit a job through an API, see the description of the **modules** parameter in Table 2 "Request parameters" in **Creating a Batch Processing Job**.

Complete Example Code

```
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
```



```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SQLContext;
import org.apache.spark.sql.SaveMode;
import java.util.Arrays;

public class TestMongoSparkSql {
  public static void main(String[] args) {
    SparkContext sparkContext = new SparkContext(new SparkConf().setAppName("datasource-mongo"));
    JavaSparkContext javaSparkContext = new JavaSparkContext(sparkContext);
    SQLContext sqlContext = new SQLContext(javaSparkContext);

//    // Read json file as DataFrame, read csv / parquet file, same as json file distribution
//    DataFrame dataFrame = sqlContext.read().format("json").load("filepath");

    // Read RDD in JSON format to create DataFrame
    JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList("{\"id\":\"5\",\"name\":\"Ann
\",\"age\":\"23\"}"));
    Dataset<Row> dataFrame = sqlContext.read().json(javaRDD);

    String url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin";
    String uri = "mongodb://username:pwd@host:8635/db";
    String user = "rwuser";
    String database = "test";
    String collection = "test";
    String password = "######";

    dataFrame.write().format("mongo")
            .option("url",url)
            .option("uri",uri)
            .option("database",database)
            .option("collection",collection)
            .option("user",user)
            .option("password",password)
            .mode(SaveMode.Overwrite)
            .save();

    sqlContext.read().format("mongo")
            .option("url",url)
            .option("uri",uri)
            .option("database",database)
            .option("collection",collection)
            .option("user",user)
            .option("password",password)
            .load().show();
    sparkContext.stop();
    javaSparkContext.close();
  }
}
```

4.4 Spark Jar Jobs Using DEW to Acquire Access Credentials for Reading and Writing Data from and to OBS

Scenario

To write the output data of a Spark Jar job to OBS, AK/SK is required for accessing OBS. To ensure the security of AK/SK data, you can use Data Encryption Workshop (DEW) and Cloud Secret Management Service (CSMS) for unified management of AK/SK, effectively avoiding sensitive information leakage and business risks caused by hard-coded or plaintext configuration of programs.

This section walks you through on how a Spark Jar job acquires an AK/SK to read and write data from and to OBS.

Prerequisites

- A shared secret has been created on the DEW console and the secret value has been stored. For details, see **Creating a Shared Secret**.
- An agency has been created and authorized for DLI to access DEW. The agency must have been granted the following permissions:
  - Permission of the **ShowSecretVersion** interface for querying secret versions and secret values in DEW: **csms:secretVersion:get**.
  - Permission of the **ListSecretVersions** interface for listing secret versions in DEW: **csms:secretVersion:list**.
  - Permission to decrypt DEW secrets: **kms:dek:decrypt**

    For details about agency permission examples, see **Customizing DLI Agency Permissions** and **Agency Permission Policies in Common Scenarios**.
- DEW can be used to manage access credentials only in Spark 3.3.1 (Spark general queue scenario) or later. When creating a Spark job, select version 3.3.1 and configure the information of the agency that allows DLI to access DEW for the job.

  For details about how to create a custom agency and configure it, see **Customizing DLI Agency Permissions**.
- To use this function, you need to configure AK/SK for all OBS buckets.

Syntax

On the Spark Jar job editing page, set **Runtime Configuration** as needed. The configuration information is as follows:

Different OBS buckets use different AK/SK authentication information. You can use the following configuration method to specify the AK/SK information based on the bucket. For details about the parameters, see **Table 4-22**.

```
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key= USER_AK_CSMS_KEY
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key= USER_SK_CSMS_KEY
spark.hadoop.fs.obs.security.provider = com.dli.provider.UserObsBasicCredentialProvider
spark.hadoop.fs.dew.csms.secretName= CredentialName
spark.hadoop.fs.dew.endpoint=ENDPOINT
spark.hadoop.fs.dew.csms.version=VERSION_ID
spark.hadoop.fs.dew.csms.cache.time.second =CACHE_TIME
spark.dli.job.agency.name=USER_AGENCY_NAME
```


Parameter Description

Table 4-22 Parameters

Parameter	Mandatory	Default Value	Data Type	Description
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key	Yes	None	String	*USER_BUCKET_NAME* needs to be replaced with the user's OBS bucket name. The value of this parameter is the key defined by the user in the CSMS shared secret. The value corresponding to the key is the user's access key ID (AK). The user must have the permission to access the bucket on OBS.
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key	Yes	None	String	*USER_BUCKET_NAME* needs to be replaced with the user's OBS bucket name. The value of this parameter is the key defined by the user in the CSMS shared secret. The value corresponding to the key is the user's secret access key (SK). The user must have the permission to access the bucket on OBS.
spark.hadoop.fs.obs.security.provider	Yes	None	String	OBS AK/SK authentication mechanism, which uses DEW-CSMS' secret management to obtain the AK and SK for accessing OBS. The default value is **com.dli.provider.UserObsBasicCredentialProvider**.
spark.hadoop.fs.dew.csms.secretName	Yes	None	String	Name of the shared secret in DEW's secret management. Configuration example: **spark.hadoop.fs.dew.csms.secretName=secretInfo**
spark.hadoop.fs.dew.endpoint	Yes	None	String	Endpoint of the DEW service to be used. See **Regions and Endpoints**. Configuration example: **spark.hadoop.fs.dew.endpoint=kms.cn-xxxx.myhuaweicloud.com**

Parameter	Mandatory	Default Value	Data Type	Description
spark.hadoop.fs.dew.csms.version	No	Latest version	String	Version number (certificate version identifier) of the common credential created in DEW's credential management. If not specified, the latest version of the common credential is obtained by default. Configuration example: **spark.hadoop.fs.dew.csms.version=v1**
spark.hadoop.fs.dew.csms.cache.time.second	No	3600	Long	Cache duration after the CSMS shared secret is obtained during Spark job access. The unit is second. The default value is 3600 seconds.
spark.hadoop.fs.dew.projectId	No	Yes	String	ID of the project DEW belongs to. The default value is the ID of the project where the Spark job is. See **Obtaining a Project ID**.
spark.dli.job.agency.name	Yes	-	String	Custom agency name.

Sample Code

This section describes how to write processed DataGen data to OBS. You need to modify the parameters in the sample Java code based on site requirements.

1. Create an agency for DLI to access DEW and complete authorization.
   For details, see **Customizing DLI Agency Permissions**.
2. Create a shared secret in DEW. For details, see **Creating a Shared Secret**.
   a. Log in to the DEW management console.
   b. In the navigation pane on the left, choose **Cloud Secret Management Service** > **Secrets**.
   c. On the displayed page, click **Create Secret**. Set basic secret information.
3. Set job parameters on the DLI Spark Jar job editing page.
   Spark Arguments

```
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key= USER_AK_CSMS_KEY
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key= USER_SK_CSMS_KEY
spark.hadoop.fs.obs.security.provider=com.dli.provider.UserObsBasicCredentialProvider
spark.hadoop.fs.dew.csms.secretName=obsAkSk
spark.hadoop.fs.dew.endpoint=kmsendpoint
spark.hadoop.fs.dew.csms.version=v3
spark.dli.job.agency.name=agency
```


4. Sample Code

   For details about the sample code, see **Using Spark Jar Jobs to Read and Query OBS Data**.

4.5 Obtaining Temporary Credentials from a Spark Job's Agency for Accessing Other Cloud Services

Function

DLI provides a common interface to obtain temporary credentials from Spark job's agencies set by users during job launch. The interface encapsulates temporary credentials obtained from the job's agency in the **com.huaweicloud.sdk.core.auth.BasicCredentials** class.

- Encapsulate temporary credentials obtained from the agency in the return value of **getCredentials()** of the **com.huaweicloud.sdk.core.auth.ICredentialProvider** interface.
- The return type is **com.huaweicloud.sdk.core.auth.BasicCredentials**.
- Only AKs, SKs, and security tokens can be obtained.
- After obtaining the AK, SK, and security token, query temporary credentials by referring to **Using CSMS to Change Hard-coded Database Account Passwords**.

Notes and Constraints

- An agency can be authorized to access temporary credentials only in Spark 3.3.1 (Spark general queue scenario).
  - When creating a Spark job, select version 3.3.1.
  - The information of the agency that allows DLI to access DEW has been configured for the job. **spark.dli.job.agency.name** indicates the custom agency name.

    For details about how to create a custom agency, see **Customizing DLI Agency Permissions**.

    Note that double quotes ("") or single quotes ('') are not required when configuring parameters.
- The Spark 3.3.1 basic image has built-in **huaweicloud-sdk-core** 3.1.62.

Preparing the Environment

The information of the agency that allows DLI to access DEW has been configured for the job.

Dependency package in POM file configurations

```
<dependency>
   <groupId>com.huaweicloud.sdk</groupId>
   <artifactId>huaweicloud-sdk-core</artifactId>
   <version>3.1.62</version>
   <scope>provided</scope>
</dependency>
```


Sample Code

This section's Java sample code demonstrates how to obtain BasicCredentials and retrieve a temporary agency's AK, SK, and security token.

- **Obtaining job agency credentials for Spark Jar jobs**

```
import org.apache.spark.sql.SparkSession;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.core.auth.ICredentialProvider;
import static com.huawei.dli.demo.DLIJobAgencyCredentialUtils.getICredentialProvider;
public class GetUserCredentialsSparkJar {
private static final Logger LOG = LoggerFactory.getLogger(GetUserCredentialsSparkJar.class);
public static void main(String[] args) throws Exception {
        SparkSession spark = SparkSession
                        .builder()
                        .appName("test_spark")
                        .getOrCreate();
  ICredentialProvider credentialProvider = getICredentialProvider();
  BasicCredentials basicCredentials = (BasicCredentials) credentialProvider.getCredentials();
  String ak = basicCredentials.getAk();
  String sk = basicCredentials.getSk();
  String securityToken = basicCredentials.getSecurityToken();
  LOG.info(">>" + " ak " + ak + " sk " + sk.length() + " token " + securityToken.length());
  spark.stop();
 }
}
```

- **Tool class for obtaining job agencies**

```
import com.huaweicloud.sdk.core.auth.ICredentialProvider;
import org.apache.spark.sql.SparkSession;
import java.util.ArrayList;
import java.util.List;
import java.util.ServiceLoader;
public class DLIJobAgencyCredentialUtils {
public static ICredentialProvider getICredentialProvider() {
        List<ICredentialProvider> credentialProviders = new ArrayList<>();
        ServiceLoader.load(ICredentialProvider.class, SparkSession.class.getClassLoader())
                    .iterator()
                    .forEachRemaining(credentialProviders::add);
        if (credentialProviders.size() != 1) {
           throw new RuntimeException("Failed to obtain temporary user credential");
        }
        return credentialProviders.get(0);
    }
}
```

