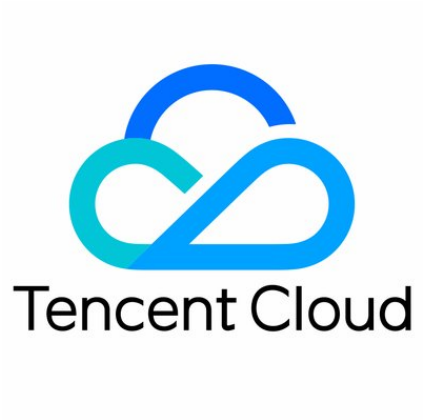
# Data Lake Compute Development Guide



Contact Us

We are committed to providing personalized pre-sales consultation and technical after-sale support. Don't hesitate to contact us at 4009100100 or 95716 for any inquiries or concerns.

# Contents

# Development Guide
# SparkJar Job Development Guide

Last updated：2024-01-10 16:42:10

## Scenarios

Data Lake Compute is fully compatible with open-source Apache Spark, allowing users to write business programs for data reading, writing, and analysis. This example demonstrates the detailed operations of reading and writing data on COS, creating databases and tables on Data Lake Compute, and reading and writing tables through Java code, assisting users in job development on Data Lake Compute.

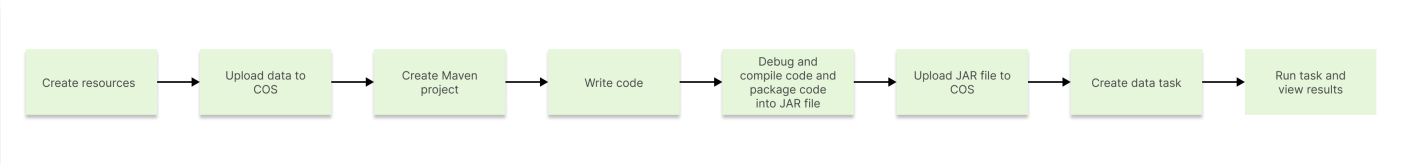
## Environment Preparation

Dependencies: JDK1.8, Maven, IntelliJ IDEA

## Development Process

### Development Flowchart

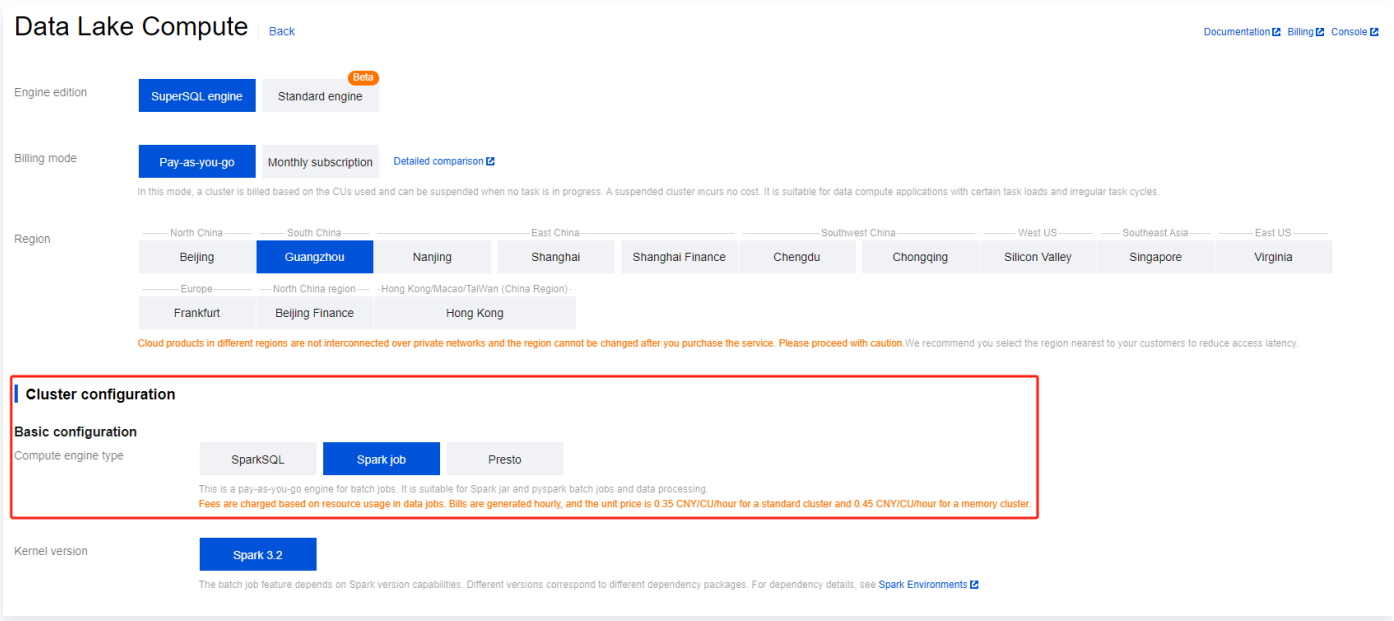
The development process for Data Lake Compute Spark JAR jobs is as follows:



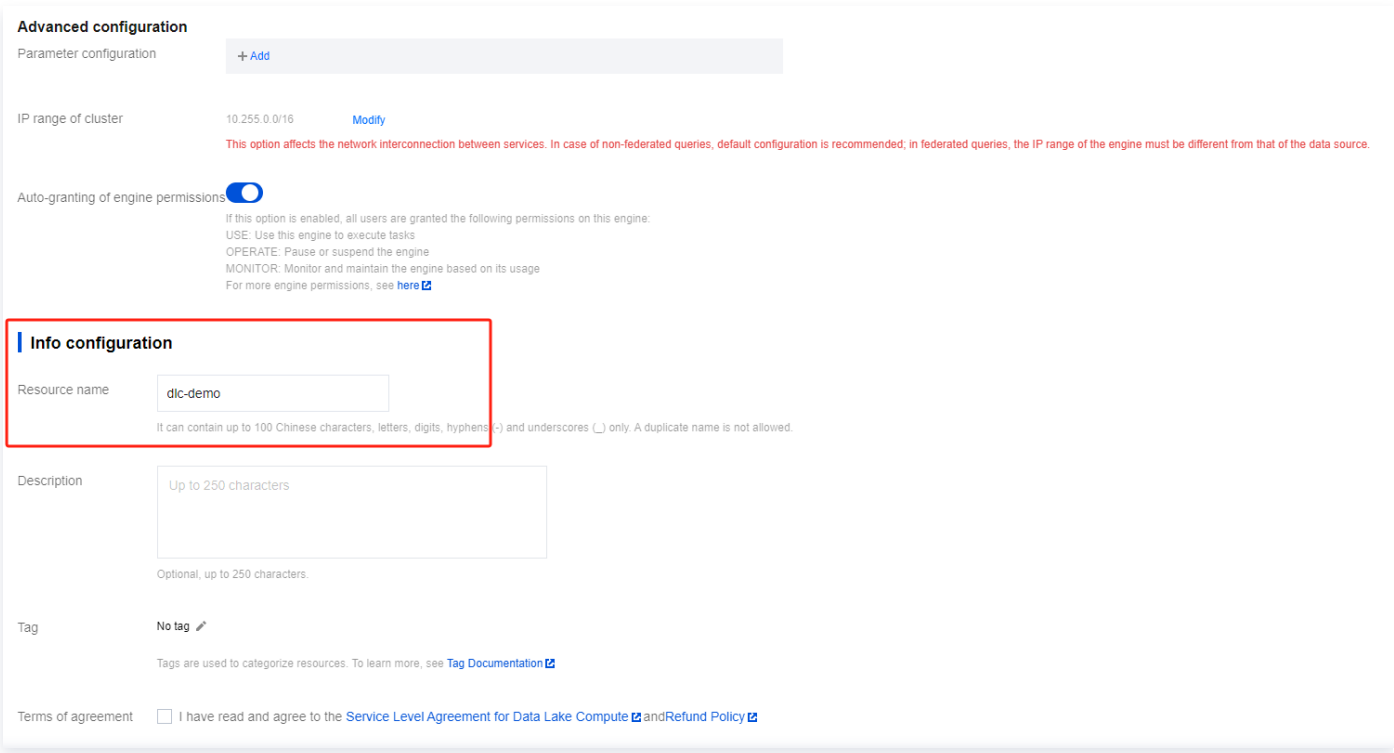

### Resource Creation

For the first time running a job on Data Lake Compute, you need to create new Spark job compute resources, for instance, creating a Spark job resource named "dlc-demo".

1. Log in to the Data Lake Compute DLC Console, select the service region, and click on **Data Engine** in the navigation menu.
2. Click **Create Resource** in the upper left corner to enter the resource configuration purchase page.
3. Select Spark as the job engine under the **Cluster Configuration > Calculation Engine Type** option.




Enter "dlc-demo" in the **Information Configuration > Resource Name** field. For more information on creating a new resource, see Purchasing a Dedicated Data Engine.




4. Click **Activate Now** to confirm the resource configuration information.
5. After ensuring the information is accurate, click **Submit** to complete the resource configuration.

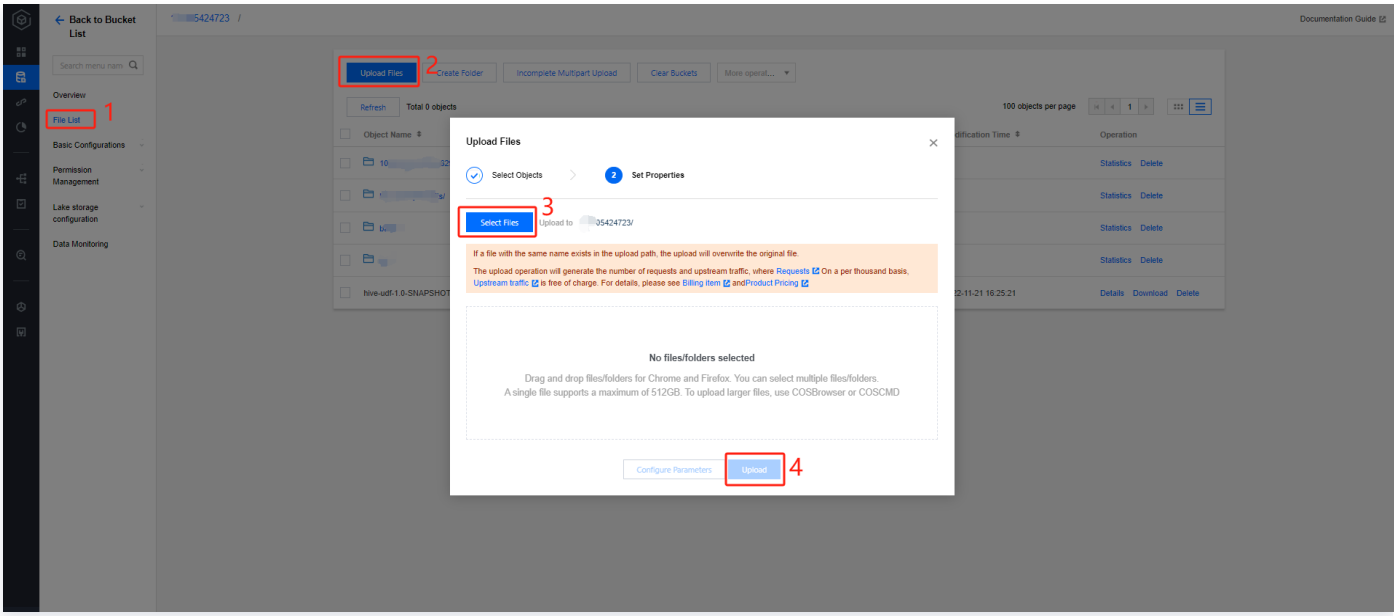
## Uploading Data to COS

Create a bucket named "dlc-demo", upload the people.json file to serve as an example for reading and writing data from COS. The content of the people.json file is as follows:

```
{"name":"Michael"}
```

```
{"name":"Andy", "age":30}
{"name":"Justin", "age":3}
{"name":"WangHua", "age":19}
{"name":"ZhangSan", "age":10}
{"name":"LiSi", "age":33}
{"name":"ZhaoWu", "age":37}
{"name":"MengXiao", "age":68}
{"name":"KaiDa", "age":89}
```

1. Log in to the Cloud Object Storage (COS) console and click on **Bucket List** in the left navigation menu.
2. Create a bucket: Click **Create Bucket** in the upper left corner, fill in the name field with "dlc-dmo", then click **Next** to complete the configuration.
3. Upload File: Click on **File List > Upload File**, select the local "people.json" file to upload to the "dlc-demo-1305424723" bucket (-1305424723 is a random string generated by the platform when creating the bucket), click **Upload** to complete the file upload. For details on creating a new bucket, refer to Create Bucket.




## Creating a Maven Project

1. Create a new Maven project named "demo" using IntelliJ IDEA.
2. Add Dependencies: Incorporate the following dependencies into the pom.xml file:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.12</artifactId>
  <version>3.2.1</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.12</artifactId>
  <version>3.2.1</version>
```

```
  <scope>provided</scope>
</dependency>
```

## Writing Code

The code functionality involves reading and writing data from COS, creating databases and tables on Data Lake Compute, and querying and writing data.

1. Example code for reading and writing data from COS:

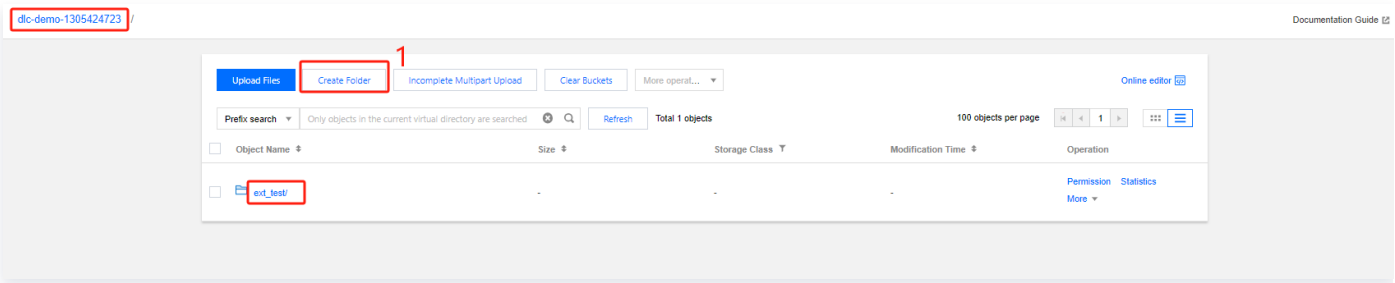
```
package com.tencent.dlc;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.SparkSession;


public class CosService {

    public static void main( String[] args )
    {
        //1. Create SparkSession
        SparkSession spark = SparkSession
                .builder()
                .appName("Operate data on cos")
                .config("spark.some.config.option", "some-value")
                .getOrCreate();
        //2. Generate a dataset by reading the JSON file on COS. It supports various types of files such as
JSON, CSV, Parquet, ORC, Text.
        String readPath = "cosn://dlc-demo-1305424723/people.json";
        Dataset<Row> readData = spark.read().json(readPath);
        //3. Perform business calculations on the dataset to generate result data. The calculations
support both API and SQL formats. Here, a temporary table is created to read data using SQL.
        readData.createOrReplaceTempView("people");
        Dataset<Row> result = spark.sql("SELECT * FROM people where age > 3");
        //4. Save the result data to COS
         String writePath = "cosn://dlc-demo-1305424723/people_output";
        // Writing supports various file types, such as JSON, CSV, Parquet, ORC, and Text.
        result.write().mode(SaveMode.Append).json(writePath);
        spark.read().json(writePath).show();
        //5. Close session
        spark.stop();
    }
}
```

2. Creating databases, tables, querying data, and writing data on Data Lake Compute:

```
package com.tencent.dlc;

import org.apache.spark.sql.SparkSession;
```

```
public class DbService {

    public static void main(String[] args) {
        //1. Initialize SparkSession
        SparkSession spark = SparkSession
                .builder()
                .appName("Operate DB Example")
                .getOrCreate();
        //2. Create Database
        String dbName = " DataLakeCatalog.dlc_db_test ";
        String dbSql = "CREATE DATABASE IF NOT EXISTS" + dbName + " COMMENT 'demo test'";
        spark.sql(dbSql);
        //3. Create an internal table
        String tableName = "test";
        String tableSql = "CREATE TABLE IF NOT EXISTS " + dbName + "." + tableName
                + "(id int, name string, age int)";
        spark.sql(tableSql);
        //4. Write Data
        spark.sql("INSERT INTO " + dbName + "." + tableName + "VALUES (1,'Andy',12),(2,'Justin',3) ");
        //5. Query Data
        spark.sql(" SELECT * FROM " + dbName + "." + tableName).show();
        //6. Creating an External Table
        String extTableName = "ext_test";
        spark.sql(
                "CREATE EXTERNAL TABLE IF NOT EXISTS " + dbName + "." + extTableName + ""
                        + " (id int, name string, age int) "
                        + "ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe' "
                        + "STORED AS TEXTFILE LOCATION 'cosn://dlc-demo-1305424723/ext_test '");
        //7. Write data to the external table
        spark.sql("INSERT INTO " + dbName + "." + extTableName + "VALUES (1,'LiLy',12),(2,'Lucy',3) ");
        //8. Query data from foreign table
        spark.sql(" SELECT * FROM " + dbName + "." + extTableName).show();
        //9. Close Session
        spark.stop();
    }
}
```

When creating an external table, follow the **steps to upload data to COS** and first create a corresponding table name folder in the bucket to save the table files.




## Debug, compile the code and package it into a JAR file

Compile and package the demo project using IntelliJ IDEA, generating the JAR package demo-1.0-SNAPSHOT.jar in the project's target folder.

## Upload the JAR Package to COS

Log in to the COS console and follow the steps in **Uploading Data to COS** to upload demo-1.0-SNAPSHOT.jar to COS.

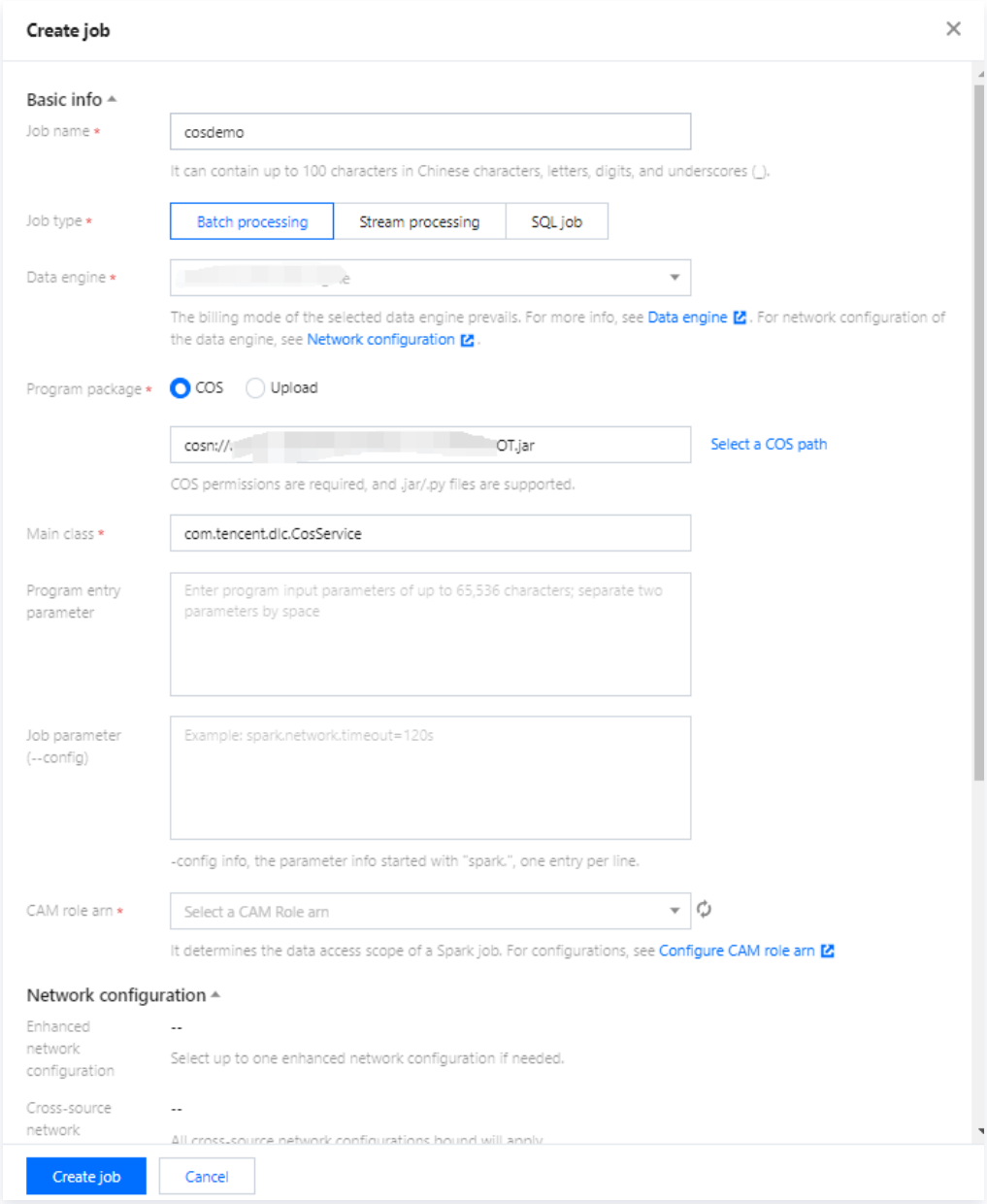
## Create a New Spark Jar Data Job

Before creating a data job, you need to complete the data access policy configuration to ensure that the data job can securely access the data. For details on configuring the data access policy, please refer to Configuring Data Access Policy. If the data policy has been configured, the name is: qcs::cam::uin/100018379117:roleName/dlc-demo.

1. Log in to the Data Lake Compute DLC Console, select the service region, and click on **Data Jobs** in the navigation menu.
2. Click **Create Job** in the upper left corner to navigate to the creation page.
3. On the job configuration page, set the job running parameters as detailed below:

| Parameter Configuration | Note |
| --- | --- |
| Job name | Customize the Spark JAR job name, for instance: cosn-demo |
| Job type | Select **Batch Processing Type** |
| Data engine | Select the dlc-demo compute engine created in the **Create Resource** step. |
| Application Package | Select COS and upload the JAR package demo-1.0-SNAPSHOT.jar in the **Upload JAR package to COS** step. |
| Main Class | Fill in according to the program code, as follows:<br>• Reading and Writing Data from COS: com.tencent.dlc.CosService<br>• For creating databases, tables, etc. on Data Lake Compute, use: com.tencent.dlc.DbService |
| CAM role arn | Select the policy created in the previous step: qcs::cam::uin/100018379117:roleName/dlc-demo |

Retain the default values of other parameters.



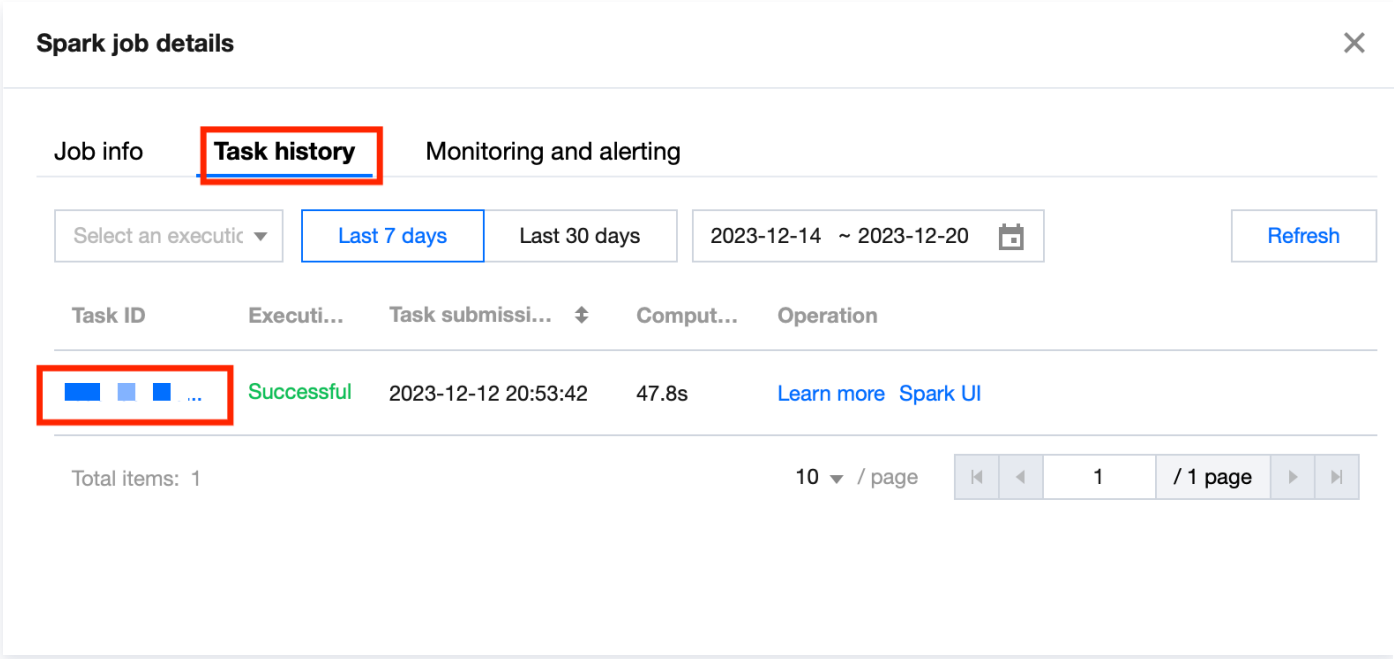

4. Click **Save** to view the created job on the **Spark Job** page.

## Execute and View Job Results

1. Run the job: On the **Spark Job** page, locate the newly created job and click **Run** to execute the job.
2. Viewing Job Execution Results: You can view the job execution logs and results.

### Viewing Job Execution Logs

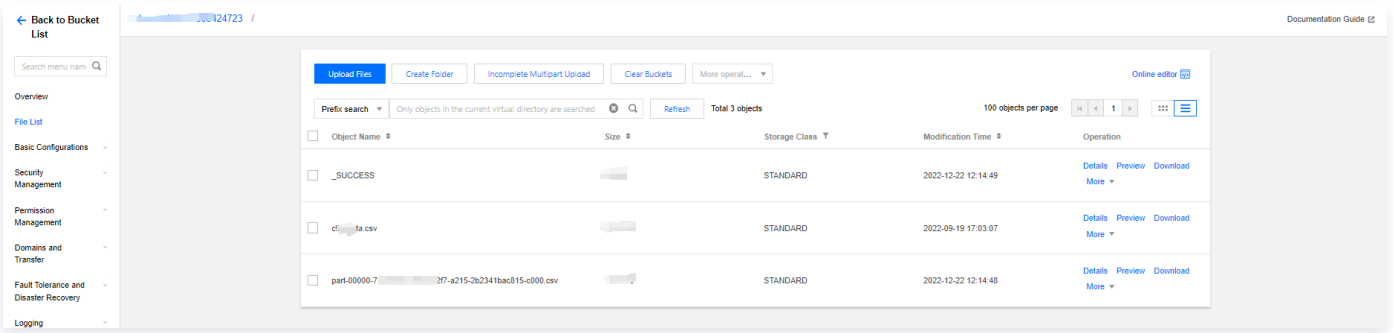
1. Click **Job Name > Historical Tasks** to view the status of the task execution.



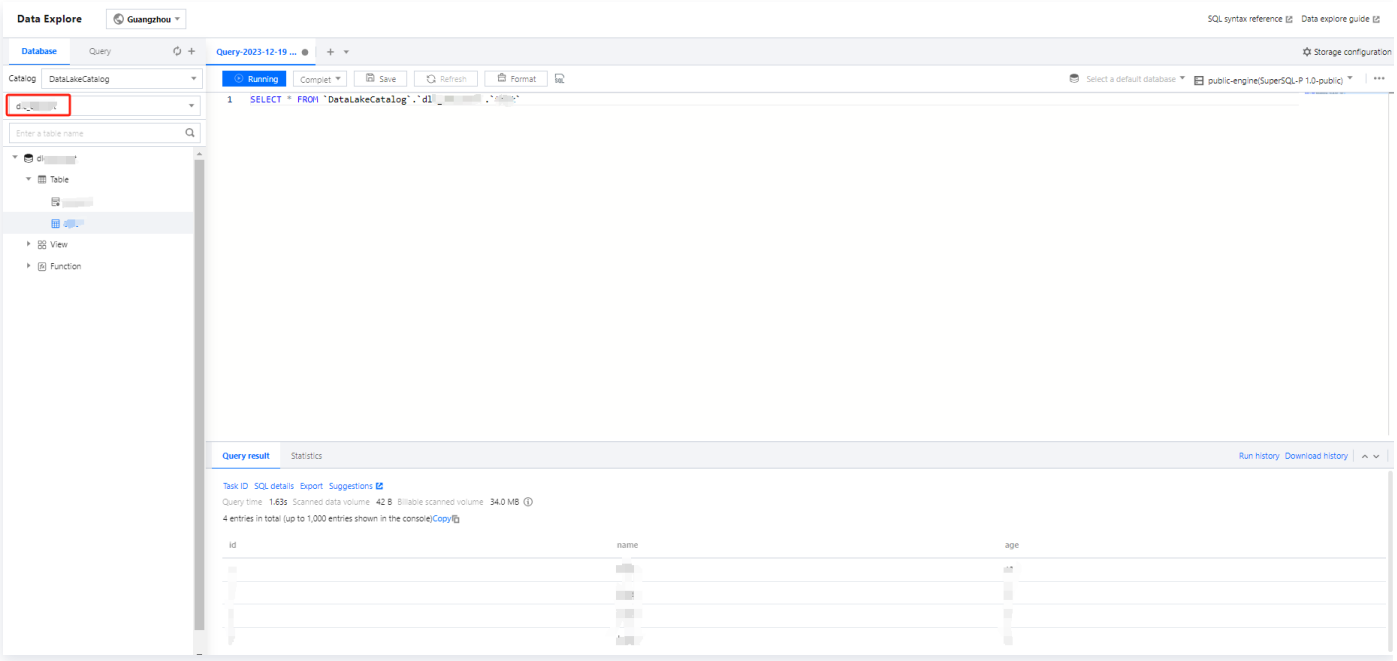

2. Click **Task ID > Run Log** to view the job execution log.

## Viewing Job Execution Results

1. To run the example of reading and writing data from COS, go to the COS Console to view the results of the data write operation.




2. To create tables and databases on Data Lake Compute, go to the **Data Exploration** page on Data Lake Compute.



# PySpark Job Development Guide

Last updated: 2024-01-10 16:42:16

## Scenarios

Data Lake Compute supports the execution of programs written in Python. This example demonstrates the detailed operations of reading and writing data on Cloud Object Storage (COS), creating libraries and tables on Data Lake Compute, and reading and writing tables, assisting users in job development on Data Lake Compute.

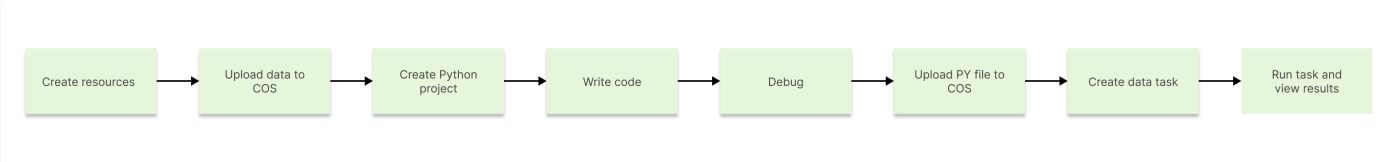
## Environment Preparation

Dependencies: PyCharm or other Python programming development tools.

## Development Process

### Development Flowchart

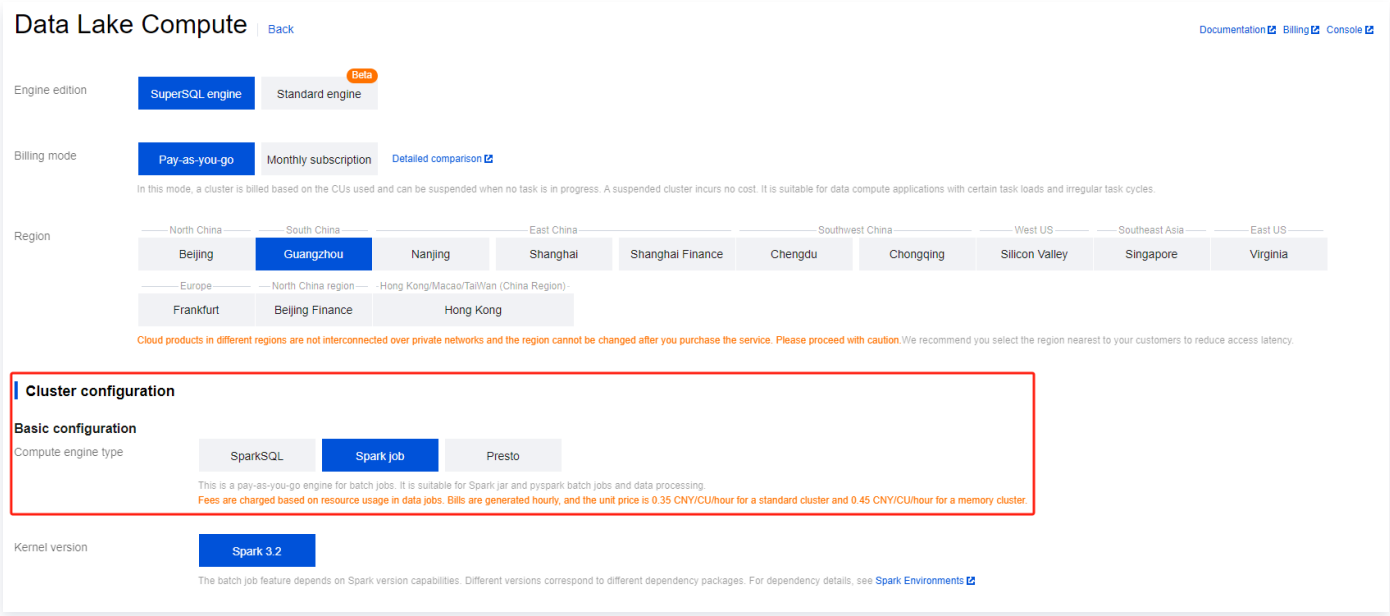
The development process for Data Lake Compute Spark JAR jobs is as follows:



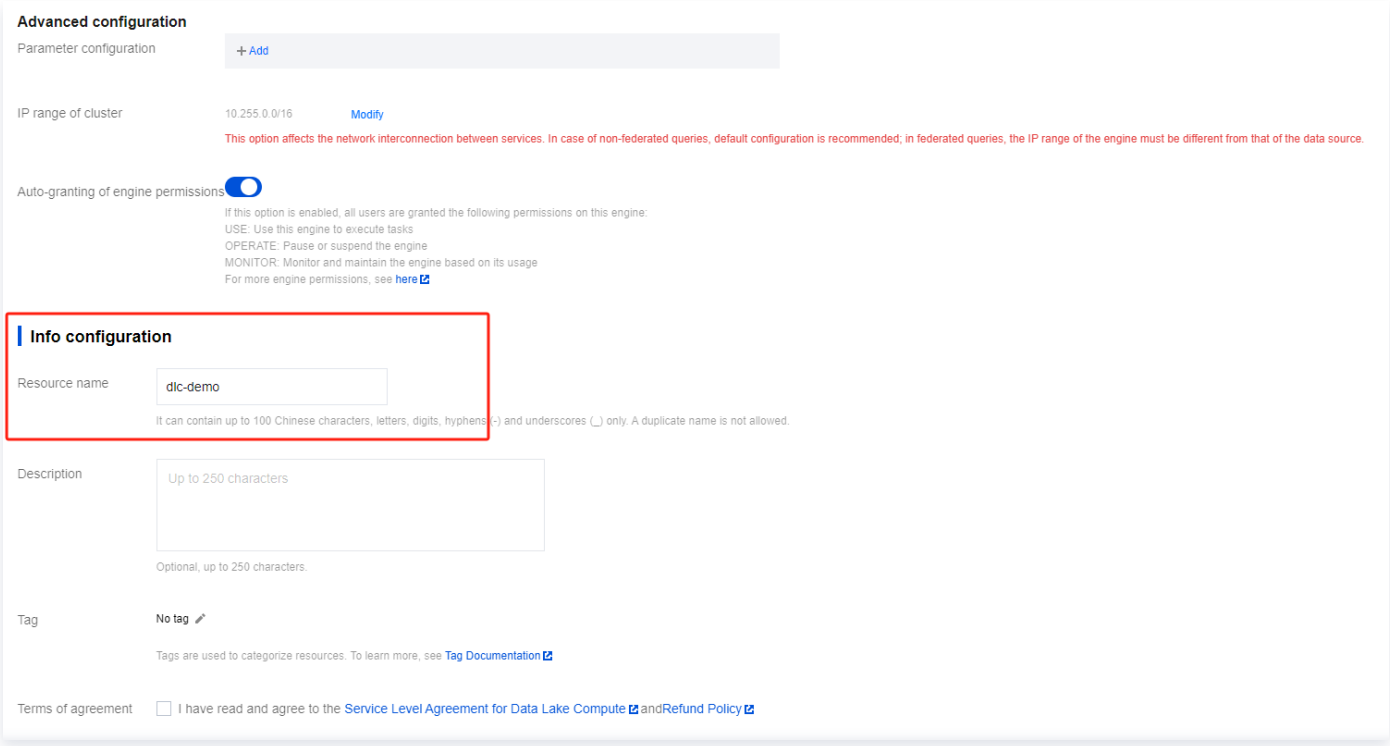

### Resource Creation

For the first time running a job on Data Lake Compute, you need to create new Spark job compute resources, for instance, creating a Spark job resource named "dlc-demo".

1. Log in to the Data Lake Compute DLC Console, select the service region, and click on **Data Engine** in the navigation menu.
2. Click **Create Resource** in the upper left corner to enter the resource configuration purchase page.
3. In the **Cluster Configuration > Calculation Engine Type** option, select Spark as the job engine.



Fill in "dlc-demo" for **Information Configuration > Resource Name**. For a detailed introduction to creating new resources, please refer to Purchasing a Dedicated Data Engine.




4. Click **Activate Now** to confirm the resource configuration information.
5. Upon verifying that the information is accurate, click **Submit** to complete the resource configuration.

## Uploading Data to COS

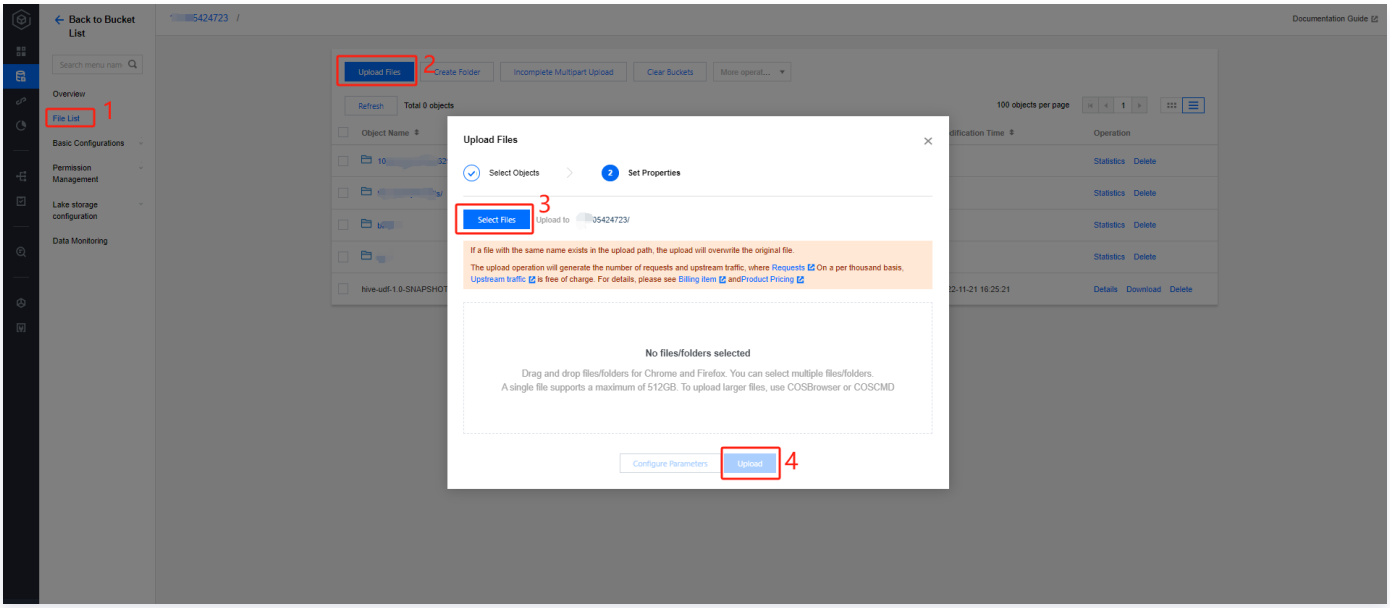
Create a bucket named "dlc-demo" and upload the file people.json for the example of reading and writing data from COS. The content of the people.json file is as follows:

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":3}
{"name":"WangHua", "age":19}
{"name":"ZhangSan", "age":10}
{"name":"LiSi", "age":33}
{"name":"ZhaoWu", "age":37}
{"name":"MengXiao", "age":68}
{"name":"KaiDa", "age":89}
```

1. Log in to the Cloud Object Storage (COS) console and click on **Bucket List** in the left navigation menu.
2. Creating a Bucket:

   Click **Create Bucket** in the upper left corner, fill in the name field with "dlc-dmo", and click **Next** to complete the configuration.
3. Upload File:

   Click on **File List > Upload File**, select the local "people.json" file to upload to the "dlc-demo-1305424723" bucket (-1305424723 is a random string generated by the platform when creating the bucket), click **Upload** to complete the file upload. For details on creating a new bucket, please refer to Create Bucket.




## Creating a Python Project

Create a new project named "demo" using PyCharm.

## Writing Code

1. Create a new cos.py file, write code with the functionality to read and write data from COS, create libraries and tables on DLC, query data, and write data.

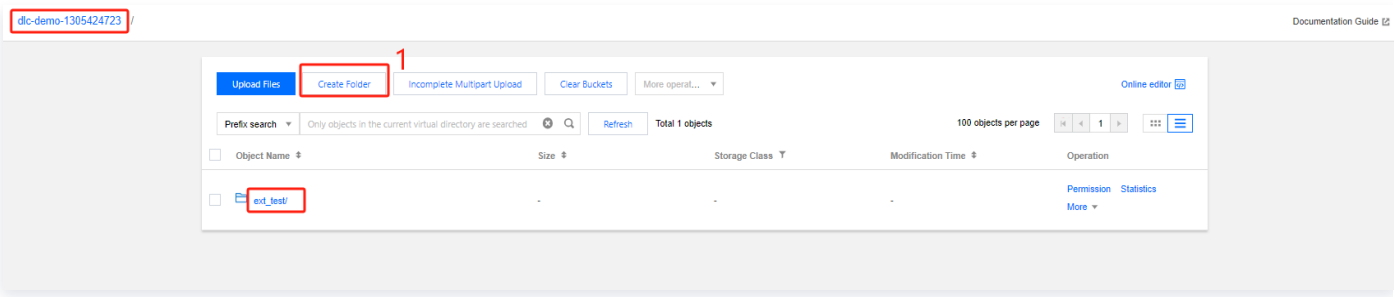
```
import sys
from pyspark.sql import SparkSession
from pyspark.sql import Row

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("Operate data on cos")\
        .getOrCreate()

    # 1. Read data from COS, supporting various file types such as JSON, CSV, Parquet, ORC, Text.
    read_path = "cosn://dlc-demo-1305424723/people.json"
    peopleDF = spark.read.json(read_path)

    # 2. Operate on the data
    peopleDF.createOrReplaceTempView("people")
    data_src = spark.sql("SELECT * FROM people WHERE age BETWEEN 13 AND 19")
    data_src.show()

    # 3. Writing Data
    write_path = "cosn://dlc-demo-1305424723/people_output"
    data_src.write.csv(path=write_path, header=True, sep=",", mode='overwrite')

    spark.stop()
```

2. Create a new db.py file, write code, the functions of which include creating libraries, tables, querying data,

and writing data on Data Lake Compute.

```
from os.path import abspath

from pyspark.sql import SparkSession



if __name__ == "__main__":

    spark = SparkSession \
        .builder \
        .appName("Operate DB Example") \
        .getOrCreate()

    # 1. Create a Database
    spark.sql("CREATE DATABASE IF NOT EXISTS DataLakeCatalog.dlc_db_test_py COMMENT 'demo test' ")
    # 2. Create Internal Table
    spark.sql("CREATE TABLE IF NOT EXISTS DataLakeCatalog.dlc_db_test_py.test(id int,name string,age int) ")
    # 3. Writing Internal Data
    spark.sql("INSERT INTO DataLakeCatalog.dlc_db_test_py.test VALUES (1,'Andy',12),(2,'Justin',3) ")
    # 4. Inspect Internal Data
    spark.sql("SELECT * FROM DataLakeCatalog.dlc_db_test_py.test ").show()

    # 5. Create External Table
    spark.sql("CREATE EXTERNAL TABLE IF NOT EXISTS DataLakeCatalog.dlc_db_test_py.ext_test(id int, name string, age int) ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe' STORED AS TEXTFILE LOCATION 'cosn://cry-1305424723/ext_test' ")
    # 6. Write external data
    spark.sql("INSERT INTO DataLakeCatalog.dlc_db_test_py.ext_test VALUES (1,'Andy',12),(2,'Justin',3) ")
    # 7. Inspect External Data
    spark.sql("SELECT * FROM DataLakeCatalog.dlc_db_test_py.ext_test ").show()
    spark.stop()
```

When creating an external table, you can follow the **steps to upload data to COS** and first create a corresponding table name folder in the bucket to save the table files.



## Debugging

Ensure PyCharm debugging is free of syntax errors.

## Upload PY Files to COS

Log in to the COS console and follow the steps in the previous section **Upload data to COS** to upload cos.py and db.py to COS.

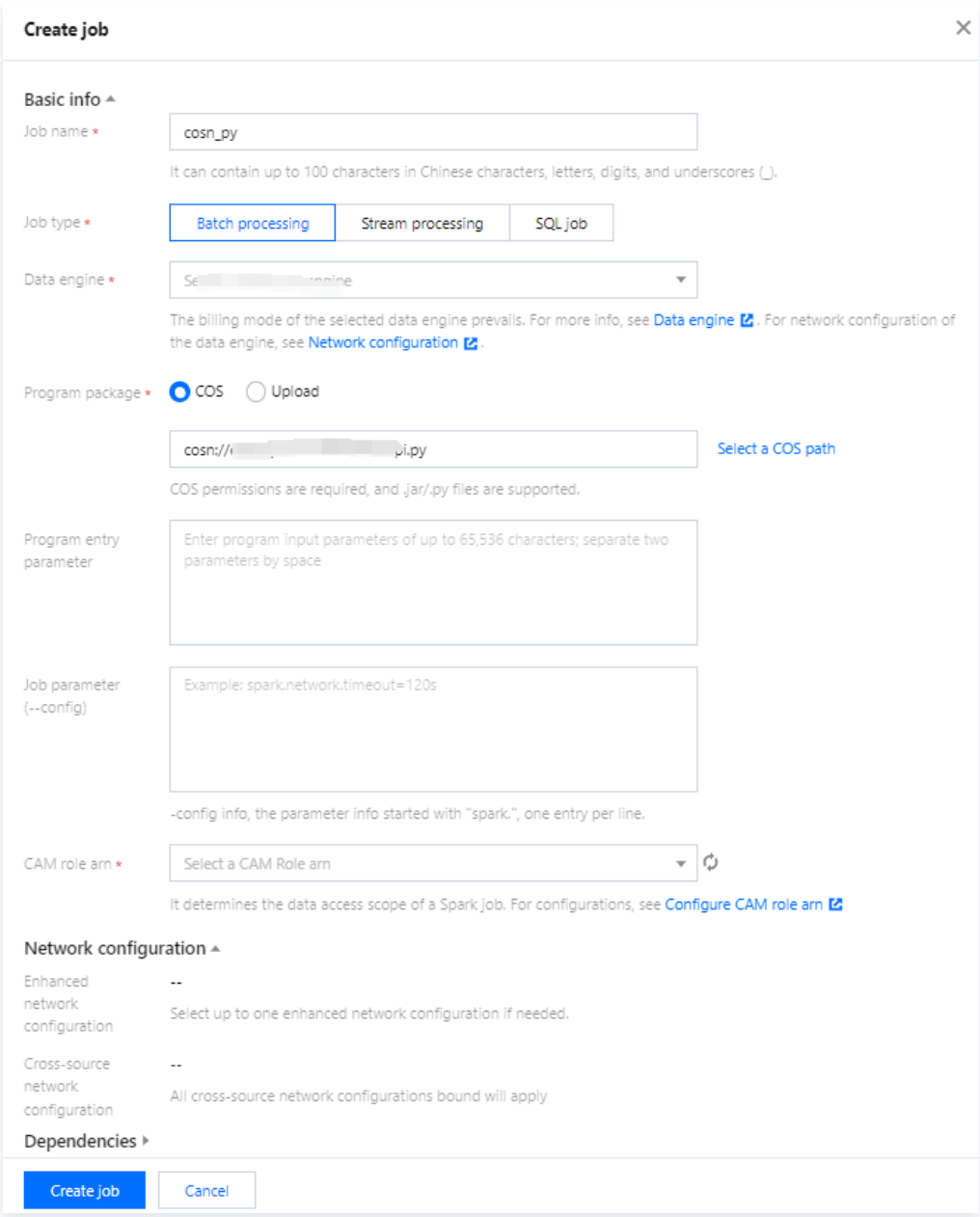
## Create a New Spark Jar Data Job

Before creating a data job, you need to complete the data access policy configuration to ensure that the data job can safely access the data. For details on configuring the data access policy, please refer to Configuring Data Access Policy. If the data policy name has been configured as: qcs::cam::uin/100018379117:roleName/dlc-demo

1. Log in to the Data Lake Compute DLC Console, select the service region, and click on **Data Jobs** in the navigation menu.
2. Click the **Create Job** button in the upper left corner to enter the creation page.
3. On the job configuration page, set the job running parameters as detailed below:

| Parameter Configuration | Note |
| --- | --- |
| Job name | Specify a custom Spark job name, for instance: cosn_py |
| Job type | Select **Batch Processing Type** |
| Data engine | Select the dlc-demo compute engine created in the **Create Resource** step. |
| Application Package | Select COS, and in the step of **uploading a py file to COS**, upload the py file:<br>• To read and write data from COS, select: cosn://dlc-demo-1305424723/cos.py<br>• To create a library, table, etc. on Data Lake Compute, select: cosn://dlc-demo-1305424723/db.py |
| CAM role arn | Select the policy created in the previous step: qcs::cam::uin/100018379117:roleName/dlc-demo |

Retain the default values of other parameters.



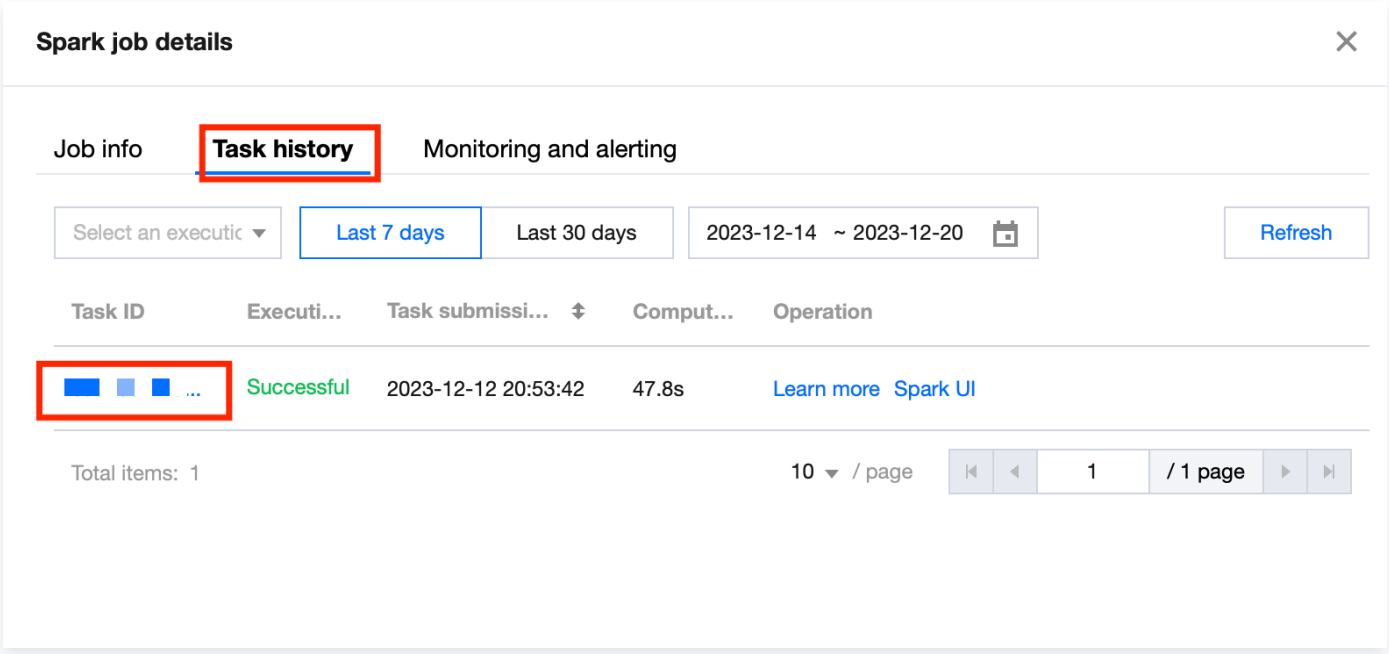

4. Click **Save** to view the created job on the **Spark Job** page.

## Execute and View Job Results

1. Run the job: On the **Spark Job** page, locate the newly created job and click **Run** to execute the job.
2. Viewing Job Execution Results: You can view the job execution logs and results.

### Viewing Job Execution Logs

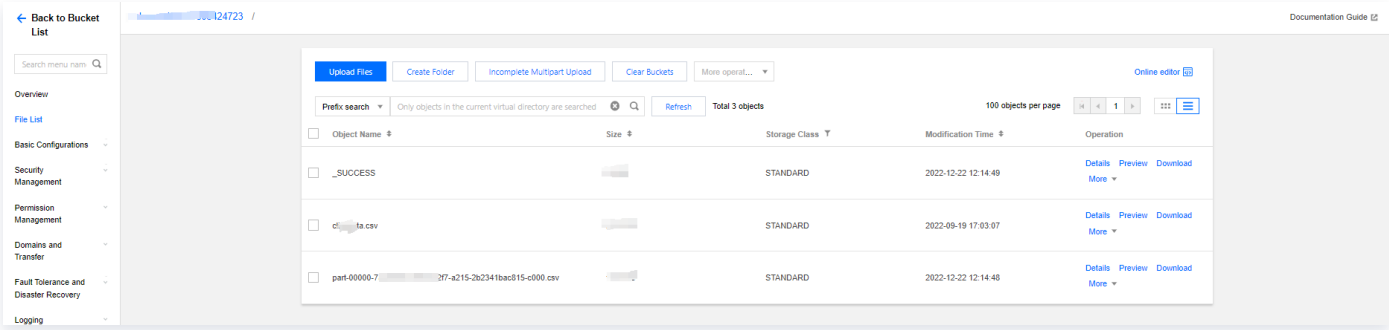
1. Click **Job Name** > **Historical Tasks** to view the task execution status:



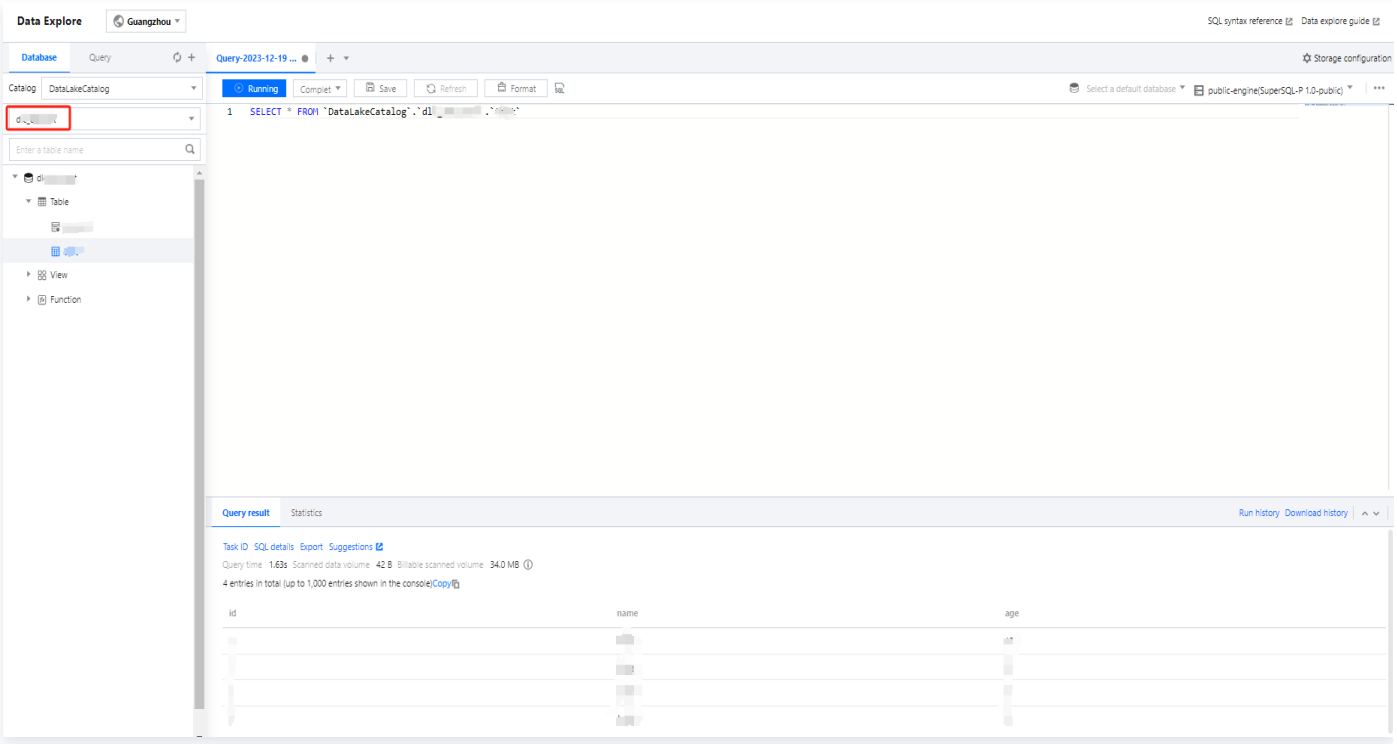

2. Click **Task ID > Run Log** to view the job execution log.

## View Job Execution Results

1. To run the example of reading and writing data from COS, go to the COS console to view the data write results.




2. To create tables and libraries on Data Lake Compute, navigate to the Data Exploration page on Data Lake Compute to view the creation of libraries and tables.



# Guide to Query Performance Optimization

Last updated: 2024-01-10 16:42:22

## Preamble

To enhance task execution efficiency, the DLC engine incorporates numerous optimization measures during computation, such as data governance, Iceberg indexing, caching, and more. Proper utilization not only reduces unnecessary scanning costs but can also boost efficiency by several folds. Herein, we provide optimization strategies across various dimensions.

## Optimizing SQL Statements

Scenario: The SQL statement itself is irrational, leading to suboptimal execution efficiency.

### Optimizing JOIN Statements

When a query involves JOIN operations across multiple tables, the Presto engine prioritizes the completion of JOIN operations on the tables on the right side of the query. Generally, executing JOIN operations on smaller tables first, followed by the result set and larger tables, enhances execution efficiency. Therefore, the order of JOIN operations directly impacts query performance. The DLC Presto automatically collects statistical data from the inner tables and uses CBO to reorder the tables in the query.
For external tables, users can typically gather statistical data through the analyze statement or manually specify the order of JOIN. If manual specification is required, please arrange the tables in order of size, placing smaller tables on the right and larger tables on the left. For instance, if table A > B > C, the statement would be: select * from A Join B Join C. It's important to note that this doesn't guarantee efficiency improvement in all scenarios. In fact, it depends on the size of the data after the JOIN.

### Optimizing GROUP BY Statements

A rational arrangement of field order in the GROUP BY statement can enhance performance. Please sort the aggregation fields in descending order based on their cardinality. For instance:

```
// Efficient Approach
SELECT id,gender,COUNT(*) FROM table_name GROUP BY id, gender;
// Inefficient Approach
SELECT id,gender,COUNT(*) FROM table_name GROUP BY gender, id;
```

Another optimization method is to use numbers as much as possible in place of specific grouping fields. These numbers represent the position of the column names after the SELECT keyword. For example, the above SQL can be replaced in the following way:

```
SELECT id,gender,COUNT(*) FROM table_name GROUP BY 1, 2;
```

### Utilizing Approximate Aggregate Functions

For query scenarios that allow minor discrepancies, employing these approximate aggregation functions can significantly enhance query performance.
For instance, in Presto, the APPROX_DISTINCT() function can be used as a substitute for COUNT(distinct x), while in Spark, the corresponding function is APPROX_COUNT_DISTINCT. The downside of this approach is

that the approximate aggregation function has an error rate of about 2.3%.

### Utilize REGEXP_LIKE instead of multiple LIKE statements

When multiple LIKE statements are present in SQL, they can typically be replaced with regular expressions to significantly enhance execution efficiency. For instance:

```
SELECT COUNT(*) FROM table_name WHERE field_name LIKE '%guangzhou%' OR LIKE '%beijing%' OR
LIKE '%chengdu%' OR LIKE '%shanghai%'
```

Can be optimized to:

```
SELECT COUNT(*) FROM table_name WHERE regexp_like(field_name,
'guangzhou|beijing|chengdu|shanghai')
```

## Data Governance

### Applicable Scenarios for Data Governance

Scenario: Real-time writing. Flink CDC real-time writing typically employs the upsert method, which generates a substantial number of small files during the writing process. When these small files accumulate to a certain extent, it can slow down data queries, or even cause them to time out and fail.
You can view the number of table files and snapshot information through the following methods.

```
SELECT COUNT(*) FROM [catalog_name.][db_name.]table_name$files;
SELECT COUNT(*) FROM [catalog_name.][db_name.]table_name$snapshots;
```

Example:

```
SELECT COUNT(*) FROM DataLakeCatalog.db1.tb1$files;
SELECT COUNT(*) FROM DataLakeCatalog.db1.tb1$snapshots;
```

When the number of table files and snapshots is excessive, you can refer to the document Enable Data Governance to activate the data governance feature.

### Data Governance Outcomes

After enabling data governance, the query efficiency has significantly improved. For instance, the table below compares the query time before and after merging files. This experiment used a 16CU Presto, with a data volume of 14M, 2921 files, and an average file size of 0.6KB.

| Executed Statement | Should Files be Consolidated? | Files | Record Count | Query Duration | Effect |
|---|---|---|---|---|---|
| SELECT count(*) FROM tb | Not required | 2,921 | 7895 | 32s | Speed: 93% Efficiency |
| SELECT count(*) FROM tb | Supported | 1 partition | 7895 | 2s | |

## Partition

Partitioning can categorize related data based on column values with different characteristics such as time and region, significantly reducing scan volume and enhancing query efficiency. For more detailed information about DLC external table partitioning, please refer to Quick Start Guide to Partitioned Tables. The table below shows the comparison of query duration and scan volume effects between partitioned and non-partitioned scenarios in a single table with 66.6GB of data, 1.4 billion data records, and an orc data format. Here, `dt` is a partitioned field with 1837 partitions.

| Query statement | Non-partitioned | | Partition | | Duration Comparison | Comparing Scan Volumes |
|---|---|---|---|---|---|---|
| | Duration | Scanned data volume | Duration | Scanned data volume | | |
| SELECT count(*) FROM tb WHERE dt='2001-01-08' | 2.6 s | 235.9MB | 480 ms | 16.5 KB | 81% Faster | Less than 99.9% |
| SELECT count(*) FROM tb WHERE dt<'2022-01-08' AND dt>'2001-07-08' | 3.8 s | 401.6MB | 2.2s | 2.8MB | 42% faster | Less 99.3% |

As can be observed from the above table, partitioning can effectively reduce query latency and scanning volume. However, excessive partitioning may have the opposite effect, as illustrated in the table below.

| Query statement | Non-partitioned | | Partition | | Duration Comparison | Comparing Scan Volumes |
|---|---|---|---|---|---|---|
| | Duration | Scanned data volume | Duration | Scanned data volume | | |
| SELECT count(*) FROM tb | 4s | 24MB | 15s | 34.5MB | 73% slower | 30% more |

We recommend filtering partitions in your SQL statements using the WHERE keyword.

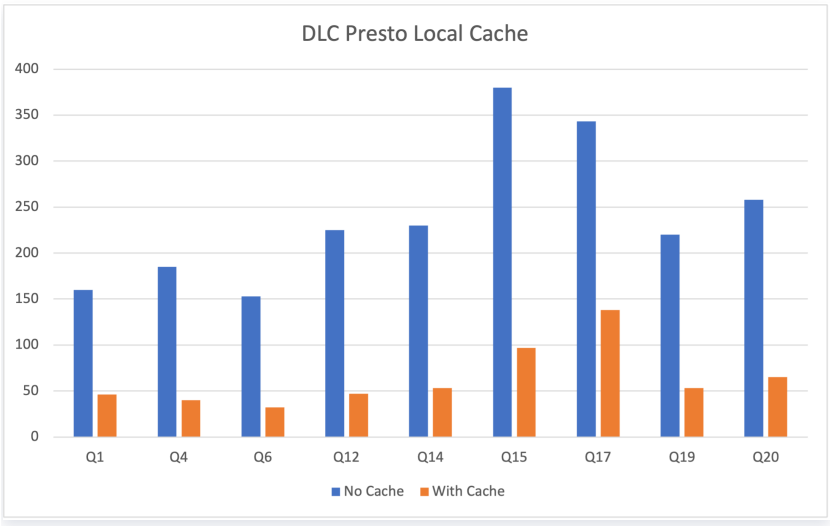
## Cache

In the current trend of distributed computing and separation of storage and computation, accessing metadata and massive data through the network will be constrained by network IO. DLC, by default, enables the following caching technologies to significantly reduce response latency, without the need for your intervention.

- Alluxio is a data orchestration technology. It provides caching, moving data from the storage layer to a location closer to data-driven applications, making it more accessible. Alluxio's memory-first hierarchical architecture allows data access speeds to be several orders of magnitude faster than existing solutions.
- RaptorX: This is a connector for Presto. It operates above the storage, offering sub-second latency, just like Presto. Its goal is to provide a unified, cost-effective, fast, and scalable solution for OLAP and interactive use cases.
- Result Caching: This technique caches repeated identical queries, significantly enhancing speed and efficiency.

The DLC Presto engine inherently supports RaptorX and Alluxio tiered caching, effectively reducing latency in similar task scenarios over a short period. Both Spark and Presto engines support result caching.

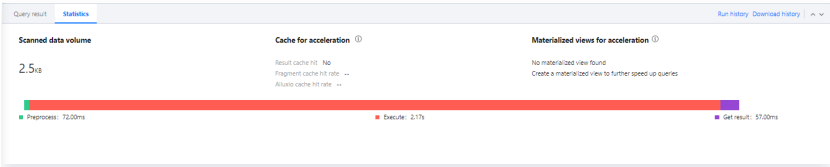
The table below presents TPCH test data in a 1TB Parquet file. This test uses 16CU Presto. As the test focuses on caching functionality, we primarily selected SQL statements with significant IO usage from TPCH. The main tables involved include lineitem, orders, customer, etc., and the SQL statements involved are Q1, Q4, Q6, Q12, Q14, Q15, Q17, Q19, and Q20. The horizontal axis represents the SQL statements, and the vertical axis represents the running time (in seconds).




It's important to note that the DLC Presto engine dynamically loads the cache based on data access frequency. Therefore, the first task execution after engine startup cannot hit the cache, resulting in the first execution still being limited by network IO. However, as the number of executions increases, this limitation is significantly alleviated. The table below shows the performance comparison of three queries in a Presto 16CU cluster.

| Query statement | Query | Duration | Data Scanning Volume |
| --- | --- | --- | --- |
| SELECT * FROM table_namewhere udid='xxx'; | Initial Query | 3.2s | 40.66MB |
| | Second Query | 2.5s | 40.66MB |
| | Third Query | 1.6s | 40.66MB |

You can view the cache hit situation of the executed SQL tasks in the Data Exploration feature of the DLC console.




## Index

Compared to external tables, the creation of internal tables with indices significantly reduces both time and scan volume. For more detailed information on table creation, please refer to Data Table Management. After creating a table, establish an index based on the frequency of business usage before the insert, following the fields indexed by WRITE ORDERED BY.

```
alter table DataLakeCatalog.dbname.tablename WRITE ORDERED BY udid;
```

The table below shows a comparison of query performance on external and internal tables (with indexing) in a Presto 16cu cluster.

| Table Type | Query | Duration | Data Scanning Volume |
| --- | --- | --- | --- |
| Outer Table | Initial Query | 16.5s | 2.42GB |
| | Second Query | 15.3s | 2.42GB |
| | Third Query | 14.3s | 2.42GB |
| Inner Table (Index) | Initial Query | 3.2s | 40.66MB |
| | Second Query | 2.5s | 40.66MB |
| | Third Query | 1.6s | 40.66MB |

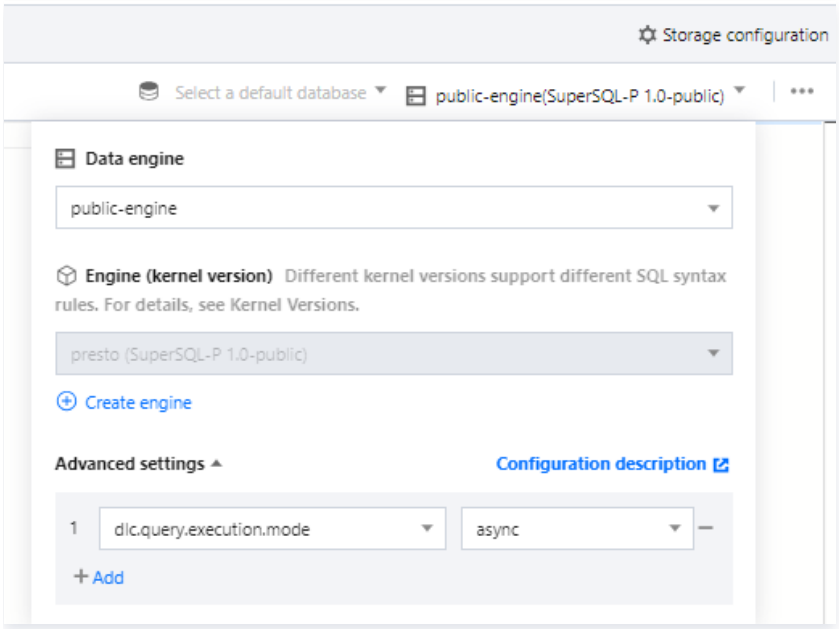
As can be observed from the table, the method of creating an internal table with an index, as compared to an external table, significantly reduces both time and scan volume. Moreover, due to cache acceleration, the execution time will also decrease with an increase in the number of executions.

## Synchronous and Asynchronous Queries

DLC has specifically optimized for BI scenarios. You can enable synchronous or asynchronous modes by configuring the engine parameter dlc.query.execution.mode (only supported by the Presto engine). The value descriptions are as follows.

- Async (default): In this mode, the task will perform a full query computation, save the results to COS, and then return them to the user. This allows users to download the query results after the query is completed.
- Sync: Under this mode, queries may not necessarily execute full computations. Once partial results are available, they are directly returned to the user by the engine without being saved to COS. Consequently, users can achieve lower query latency and duration, but the results are only retained in the system for 30 seconds. This mode is recommended for users who do not need to download the complete query results from COS but expect lower query latency and duration, such as during the query exploration phase or BI result display.

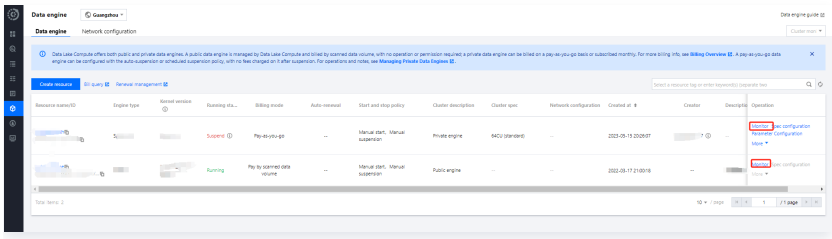
Configuration Method: After selecting the data engine, you can configure its parameters. Once the data engine is selected, click 'Add' in the advanced settings to start the configuration.

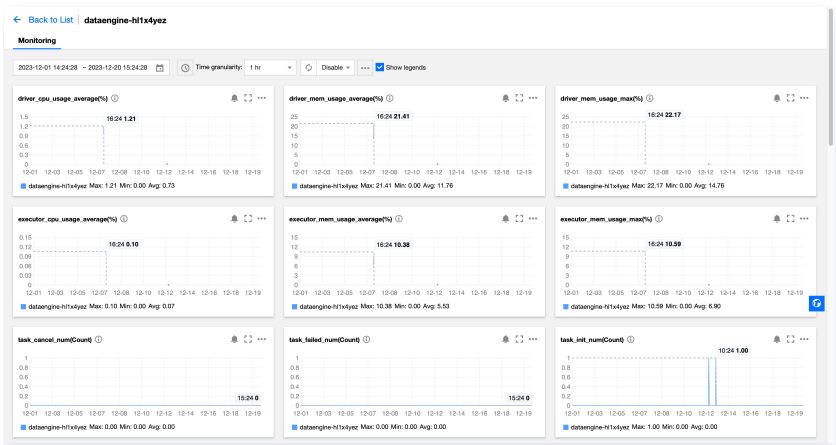



## Resource Bottlenecks

Evaluate whether resources have reached their limit. The DLC engine provides monitoring for CPU, memory, cloud disk, and network resources. You can adjust resource specifications based on the scale of your business. For configuration changes, please refer to the Configuration Adjustment Cost Description. The steps to view engine resource usage are as follows:

1. Open the Data Engine tab on the left.
2. Click the **Monitor** button on the right side of the corresponding engine.
3. Navigate to the Tencent Cloud Observability Platform to view all monitoring metrics, as shown below. For detailed operations and monitoring metrics, please refer to the Data Engine Monitoring. You can also configure alarms for each metric. For a detailed introduction, please refer to Monitoring Alarm Configuration.





# Other Factors

## Adaptive Shuffle

To enhance stability, DLC enables adaptive shuffle by default. This is a system that supports regular shuffle with limited local disk space while ensuring stability in scenarios with large shuffle and data skew. The advantages of adaptive shuffle include:

1. Reducing Storage Costs: The disk mount volume of cluster nodes is further reduced, with each node in a general scale cluster requiring only 50G, and large-scale clusters not exceeding 200G.
2. Stability: The task execution stability will no longer fail due to local disk limitations in scenarios where the shuffle data volume dramatically increases or data skew occurs.

Although adaptive shuffle reduces storage costs and enhances stability, in certain scenarios, such as when resources are insufficient, it can introduce approximately 15% latency.

## Cold Start of the Cluster

DLC supports automatic or manual suspension of clusters, which ceases any further charges. Therefore, upon the first task execution after cluster startup, there might be a "Queuing" notification due to the cold start of the cluster pulling up resources. If you frequently submit tasks, it is recommended to purchase a yearly or monthly subscription cluster. This type of cluster does not have a cold start and can execute tasks quickly at any time.

# UDF Function Development Guide

Last updated: 2024-01-10 16:42:28

## UDF description

Users can write UDF functions, package them into JAR files, and define them as functions in data lake computations for use in query analysis. Currently, the UDF of Data Lake Computing DLC is in HIVE format, inheriting org.apache.hadoop.hive.ql.exec.UDF, and implementing the evaluate method.
Example: Simple array UDF function.

```
public class MyDiff extends UDF {
    public ArrayList<Integer> evaluate(ArrayList<Integer> input) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        result.add(0, 0);
        for (int i = 1; i < input.size(); i++) {
        result.add(i, input.get(i) - input.get(i - 1));
        }
        return result;
    }
}
```

Reference for pom file:

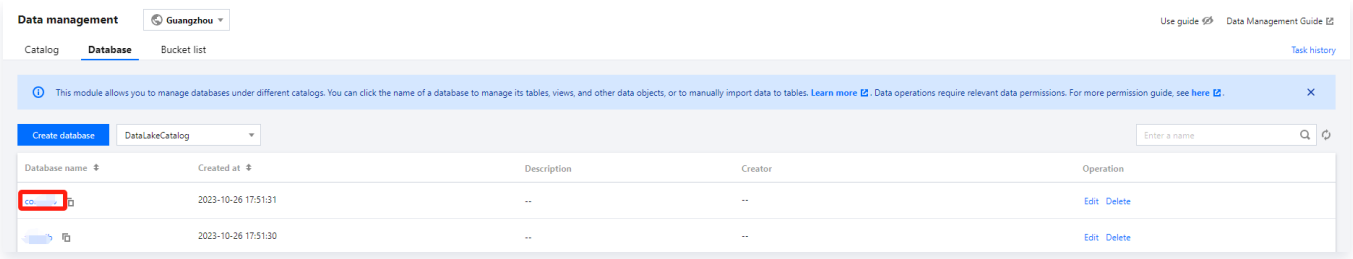
```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.16</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

## Creating function version alias

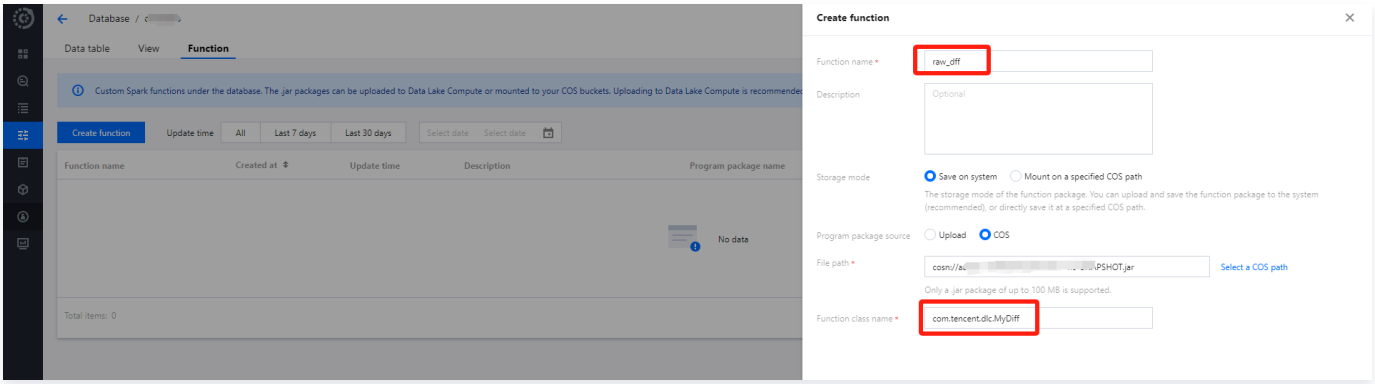
> **Note:**
> If you are creating udaf/udtf functions, you need to add the _udaf/_udtf suffix to the function name accordingly.

If you are familiar with SQL syntax, you can create a function by executing the CREATE FUNCTION syntax through **Data Exploration**, or you can create it through a visual interface, as follows:

1. Log in to the Data Lake Compute console and select the service region.
2. Navigate through the left sidebar menu to **Data Management**, select the database where you want to create the function. If you need to create a new database, refer to Database Management.



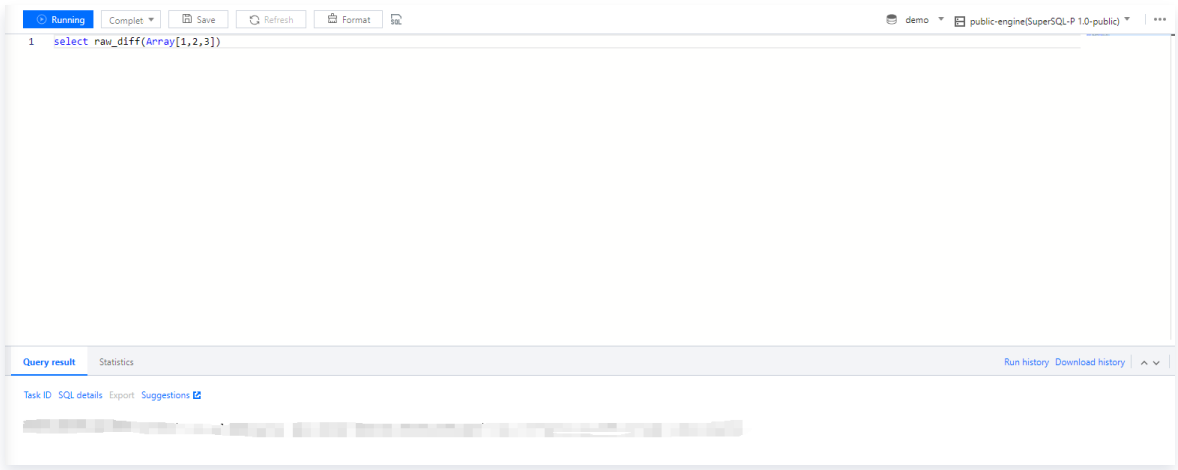

3. Click on **Functions** to navigate to the function management page.
4. Click on **Create Function** to proceed with the creation.




The UDF package supports either local upload or selection of a COS path (requires relevant COS permissions). The example here involves creating via a COS path. The function class name includes both the "package information" and the "executable class name of the function".

## Function utilization

1. Log in to the Data Lake Compute console and select the service region.
2. Navigate through the left sidebar menu to Data Exploration, select the computation engine, and you can then utilize SQL to invoke functions.



# Materialized View

Last updated: 2024-01-10 16:42:34

> **Note:**
> Currently, Data Lake Compute (DLC) materialized views only support SparkSQL and Presto engines.

A Materialized View is a unique object within a database, embodying a pre-calculated and stored query result set. It offers rapid query performance when dealing with substantial data volumes and intricate queries. While enhancing query performance, materialized views also introduce storage and computation costs. We recommend utilizing materialized views in the following scenarios:

- When the source table undergoes infrequent changes.
- When compared to the source table, the number of fields and results in the materialized view table is significantly reduced.

DLC supports both standard materialized views and mapped materialized views. The following provides an introduction and comprehensive usage examples. For a list of supported syntax, please refer to Materialized View Syntax.

## Standard Materialized View

The fundamental usage process for a standard materialized view encompasses creation, refreshment, and utilization.
The following illustrates a complete process using the Presto engine as an example.

### Data Preparation

Execute SQL to create a table and insert data. The following statement creates a table named `student`.

```
CREATE DATABASE IF NOT EXISTS mv_test3;
create table student(id int, name string, score int);
insert into student values (1,'zhangsan', 90);
insert into student values (2,'lisi', 100);
insert into student values (3,'wangwu', 80);
insert into student values (4,'zhaoliu', 30);
select * from student order by id;
```

### Creating a Standard Materialized View

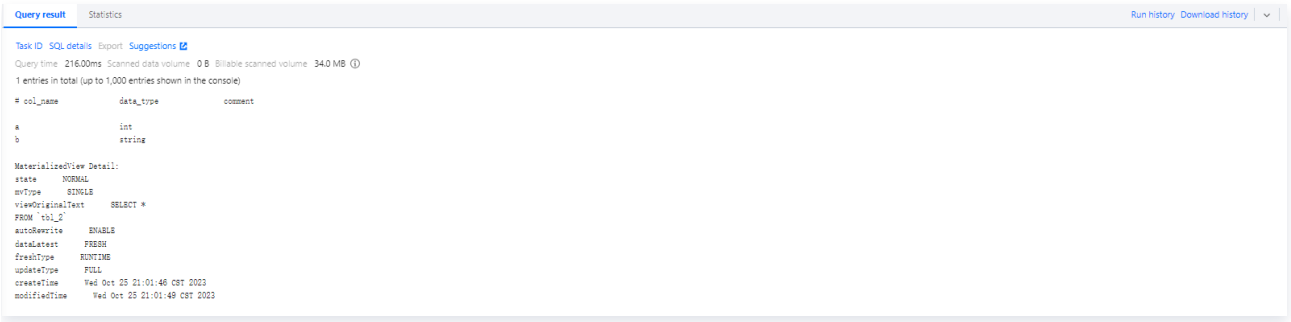
Use the `CREATE MATERIALIZED VIEW` statement to create a materialized view. Specify the name of the materialized view and the query statement, optionally designating the source table and conditions for the query. In the following example, a simple SELECT statement is used to select all scores from the table `student` and perform a sum operation on them. This sum result is then used as the content for the materialized view `mv_student_sum`.

```
CREATE MATERIALIZED VIEW mv_student_sum AS (
  select sum(score) from student
);
```

## View Materialized View Details

Use the `DESCRIBE MATERIALIZED VIEW` statement to view detailed information about the materialized view, including its name, query statement, and refresh status.

```
DESCRIBE MATERIALIZED VIEW mv_student_sum;
```




## Manually refreshing the materialized view.

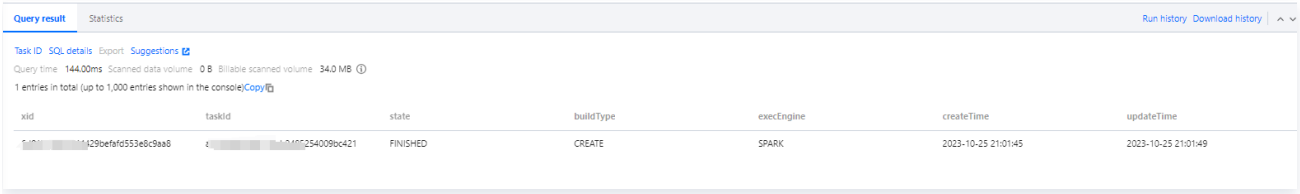
Use the `REFRESH MATERIALIZED VIEW` statement to manually refresh the data in the materialized view. This is merely a demonstration. In most cases, you do not need to manually refresh the materialized view. As long as the SQL hits a materialized view with changes in the source table, it will refresh automatically.

```
REFRESH MATERIALIZED VIEW mv_student_sum;
```

## Viewing the execution task list of materialized views.

Use the `SHOW MATERIALIZED VIEW JOBS` command to view the execution task list of the materialized view, providing insight into the refresh history and status of the materialized view.

```
SHOW MATERIALIZED VIEW JOBS IN mv_student_sum;
```




## Execution of SQL rewriting.

When querying data using SELECT statements, with the expectation of automatic rewriting and hitting the materialized view. You can check whether it has been automatically rewritten to the materialized view through the statistical data in the query results.

```
select sum(score) from student;
```

## Deleting a Materialized View

```
DROP MATERIALIZED VIEW mv_student_sum;
```

## Mapping Materialized View

A mapped materialized view is a distinct type of materialized view that associates with an existing table. By utilizing mapped materialized views, one can associate the query results of the materialized view with the data of an existing table, thereby optimizing the query performance of the existing table.

### Restrictions

Relative to standard materialized views, materialized views have the following limitations:

- Mapped materialized views do not support refresh operations, meaning that the data of the materialized view cannot be refreshed using the REFRESH MATERIALIZED VIEW statement. Consequently, the data of the materialized view can only remain consistent with the data of the mapped table and cannot be updated automatically.
- The materialized view does not automatically rewrite SQL, meaning the query statement will not automatically convert to use the materialized view. It necessitates manually specifying the query statement that utilizes the materialized view.
- When deleting a mapped materialized view, only the association with the mapped table will be removed, not the mapped table itself. The mapped table will continue to exist and can still be utilized.

### Recommended scenarios

We suggest employing mapped materialized views in the following scenarios:

- When there is an existing table with a large volume of data and its query performance is subpar, query performance can be optimized by mapping a materialized view.
- When it is necessary to maintain data consistency between the materialized view and the existing table, and there is no need for automatic refresh of the materialized view, a mapped materialized view can be utilized.

### When the source table is of the Iceberg type.

When the Iceberg table is the source table, a complete configuration sample is as follows:

#### Creating mapped materialized views based on CTAS (Create Table As Select).

The name of the materialized view to be mapped must be consistent with the table to be mapped. The following example first creates a table based on CTAS for the creation of the mapped MV. Data preparation can refer to the data preparation section in the complete example of a regular materialized view.

```
CREATE TABLE link_mv_student AS (
  select sum(score) from student
);
--Create a mapped materialized view: Use the CREATE MATERIALIZED VIEW statement to establish a
mapped materialized view.
-- When creating a materialized view, use the WITH META LINK clause and specify the name of the
mapping table as the association.
CREATE MATERIALIZED VIEW link_mv_student WITH META LINK AS (
  select sum(score) from student
);
```

#### View Mapped Materialized View

The DESCRIBE MATERIALIZED VIEW statement can be used to view detailed information about the mapped materialized view, including its name, query statement, and refresh status.

```
DESCRIBE MATERIALIZED VIEW link_mv_student;
SHOW MATERIALIZED VIEW JOBS IN link_mv_student;
```

### Mapped materialized views do not support refresh operations.

Mapped materialized views do not support REFRESH operations, meaning the data in the materialized view cannot be refreshed using the REFRESH MATERIALIZED VIEW statement. Consequently, the data in the materialized view can only remain consistent with the data in the mapped table and cannot be updated automatically.

### SQL Rewriting

The mapped materialized view will not automatically rewrite the SQL query statement.
For instance, executing `select sum(score) from student;` will not hit the mapped materialized view.
You can specify the allowance for SQL rewriting based on mapped materialized views by utilizing Hint or TaskConf parameters.

```
-- Manually specify the SQL that needs to be rewritten.
select /*+ OPTIONS('eos.sql.materializedView.enableRewrite'='true') */
sum(score) from student;
```

### Deleting Mapped Materialized Views

Utilize the DROP MATERIALIZED VIEW statement to delete the mapped materialized view. After deleting the mapped materialized view, only the association with the mapped table will be removed, while the mapped table itself will continue to exist.

```
DROP MATERIALIZED VIEW link_mv_student;
DESCRIBE link_mv_student; -- This can be used to check if the source table still exists.
```

## When dealing with Hive type source tables.

When using a Hive table as the source table, a complete example is as follows:

### Preparing to initialize data.

Initially, it is necessary to prepare the initial data and create a Hive base table. Utilize the CREATE EXTERNAL TABLE statement to establish a Hive base table, and manually insert data using the INSERT statement.

```
CREATE EXTERNAL TABLE student_2(id int, name string, score int)
LOCATION 'cosn://guangzhou-test-1305424723/mv_test4/student_2';
insert into student_2 values (1,'zhangsan', 90);
insert into student_2 values (2,'lisi', 100);
insert into student_2 values (3,'wangwu', 80);
insert into student_2 values (4,'zhaoliu', 30);
select * from student_2;
```

### Establish a mapped Hive external table.

Utilize the CREATE EXTERNAL TABLE statement to establish a mapped Hive external table.

```
CREATE EXTERNAL TABLE link_mv_student_hive (
sum_score BIGINT
) LOCATION 'cosn://guangzhou-test-1305424723/mv_test4/link_mv_student_hive';
```

Insert data into the mapping table, using the INSERT OVERWRITE statement to insert the query results into the mapping table, ensuring the data in the mapping table is consistent with the data in the Hive base table.

```
-- Inserting data into the mapped table
INSERT OVERWRITE link_mv_student_hive
select sum(score) from student;
```

### Creating mapped materialized views based on Hive external tables.

Use the CREATE MATERIALIZED VIEW statement to create a mapped materialized view. When creating a materialized view, use the WITH META LINK clause and specify the name of the aforementioned Hive external table as the association.

```
CREATE MATERIALIZED VIEW link_mv_student_hive WITH META LINK AS (
    select sum(score) from student_2
);
```

# Guide to Developing in Hudi Table Format Overview

Last updated: 2024-01-10 16:42:52

Apache Hudi is a next-generation streaming data lake platform. Its most prominent feature is the support for record-level upserts and deletions, along with incremental queries.
You can utilize the Hudi table format when creating, writing to, and querying tables. If you encounter any issues with the Hudi table format on DLC, feel free to submit a ticket to contact us.

## Scenarios

- Near Real-Time Data Ingestion
  Hudi supports the ability to insert, update, and delete data. Compared to other traditional file formats, Hudi optimizes the issue of small files generated during the data writing process.
  You can use DLC Spark or Flink to ingest log data from message queues (such as Kafka) into Hudi in real-time. It also supports real-time synchronization of change data generated by database Binlog.
- Incremental Data Processing
  In the past, incremental processing often partitioned data into hourly granules. Once the data within this partition was written, the partition could provide corresponding queries, making the "freshness" of the data reach an hourly level. However, if data latency occurs, the only remedy is to ensure accuracy through recalculating the entire partition, which increases the performance overhead of the entire system in terms of computation and storage.
  Hudi supports the Incremental Query type. You can use DLC Spark Streaming to query data changes that occurred after a given COMMIT. This reduces the consumption of computational resources and can elevate the freshness of data from an hourly level to a minute level, allowing data to flow quickly between different layers within the lake.
- Near Real-Time Data Analysis
  By reducing the data update time to a few minutes, Hudi provides a more effective solution for real-time analysis. Moreover, with the seamless integration and excellent performance of DLC Presto and SparkSQL with Hudi, you can perform faster analysis on more real-time data without any additional configuration.

## Guide to Developing in Hudi Table Format

- Create a Hudi Table
- Hudi Data Writing
- Hudi Data Query

# Create a Hudi Table

Last updated: 2024-01-10 16:42:58

## Explanation of Hudi Table Types

Hudi supports the following two types of tables:

- Copy On Write
  Abbreviated as COW, it stores data in Parquet format. The update operations for Copy On Write tables are implemented through rewriting.
- Merge On Read
  Abbreviated as MOR, it employs a hybrid approach to data storage, using both columnar (Parquet) and row-based (Avro) file formats. Merge On Read stores base data in a columnar format, while incremental data is stored in a row-based format. The most recently written incremental data is stored in row-based files, and a COMPACTION operation is performed according to a configurable strategy to merge incremental data into columnar files.

The differences between the two types of tables are illustrated in the following table.

| Trade-offs | Copy On Write | Merge On Read |
| --- | --- | --- |
| Data latency | Height | Low |
| Query Latency | Low | Height |
| Update Costs | High (rewrite the entire Parquet file) | Low (Appended to Incremental Log) |
| Write Amplification | Height | Low |
| Scenarios | Write less, read more | High write, low read, real-time upsert |

## Creating a Hudi Table with Spark SQL

### Syntax Format and Parameter Descriptions

DLC Spark engine supports the direct creation of Hudi tables using SQL. For more details, please refer to the syntax format and examples.

- Syntax Format

```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
  ( col_name[:] col_type [ COMMENT col_comment ], ... )
USING data_source
  [ COMMENT table_comment ]
  [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
  [ LOCATION path ]
  [ TBLPROPERTIES ( property_name=property_value, ... ) ]


CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  USING hudi
  [ COMMENT table_comment ]
  [ PARTITIONED BY ( col_name1, col_name2, ... ) ]
```

```
  [ LOCATION path ]
  [ TBLPROPERTIES ( property_name=property_value, ... ) ]
```

- TBLPROPERTIES Parameter Description

| Category | Default value | Description |
| --- | --- | --- |
| primaryKey | uuid | Specify the primary key column. When there are multiple primary keys, separate them with a comma. |
| type | cow | Table Type: Two types are supported as follows: COW, representing Copy-On-Write type tables, and MOR, representing Merge-On-Read type tables. |
| preCombineField | - | This value is used to merge and deduplicate rows with the same key before writing. It corresponds to the DataSourceWriteOptions.PRECOMBINE_FIELD_OPT_KEY field in Hudi. |

## Sample

- Creating a Non-Partitioned Table

```
create table hudi_mor_tbl (
  id int,
  name string,
  price double,
  ts bigint
) using hudi
comment 'hudi demo'
location 'cosn://<cos_bucket>/spark_hudi/hudi_mor_tbl'
tblproperties (
  'type' = 'mor',
  'primaryKey' = 'id',
  'preCombineField' = 'ts'
);
```

- Creating a Partitioned Table

```
create table hudi_cow_pt_tbl (
  id bigint,
  name string,
  ts bigint,
  dt string,
  hh string
) using hudi
comment 'hudi partition demo'
partitioned by (dt, hh)
location 'cosn://<cos_bucket>/spark_hudi/hudi_cow_pt_tbl'
tblproperties (
  'type' = 'cow',
  'primaryKey' = 'id',
```

```
  'preCombineField' = 'ts'
)
```

# Data Writing in Hudi

Last updated: 2024-01-10 16:43:03

DLC Hudi currently supports the use of Spark Streaming and Flink to real-time import external data into the lake.

## Real-time Writing with Spark Streaming

DLC supports the deployment of Spark Streaming jobs and recommends using DLC Spark jobs to write into DLC Hudi tables.

- Code Example

```
kafkaDF.writeStream
  .option("checkpointLocation","cosn://<cos_bucket>/spark_hudi/spark_ck")
  .trigger(Trigger.ProcessingTime(10, TimeUnit.SECONDS))
  .queryName("write hudi")
  .foreachBatch((batchDF:DataFrame,_:Long)=>{
     batchDF.write
     .mode(SaveMode.Append)
     .format("hudi")
     .option("hoodie.datasource.write.table.type","MERGE_ON_READ")
     .option("hoodie.datasource.write.precombine.field","ts")
     .option("hoodie.datasource.write.recordkey.field","uuid")
     .option("hoodie.datasource.write.partitionpath.field","partitionpath")
     .option("hoodie.datasource.write.table.name","hudi_mor")
     .save("cosn://<cos_bucket>/spark_hudi/hudi_mor")
  }).start().awaitTermination()
```

## Real-time Writing with Flink

In addition to the recommended DLC SPARK jobs, you can also opt for Tencent Cloud Stream Compute Service to write into DLC Hudi in real-time. For more details, refer to the Oceanus product documentation.
If you wish to write into DLC Hudi tables through a self-built Flink program, you can refer to the following sample code:

```
// Prepare the Flink Stream Table execution environment
EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()
    .build();
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, settings) ;
```

```
// Specify the Kafka source table
tableEnv.executeSql("CREATE TABLE tbl_kafka (\n" +
    "\tuuid STRING,\n" +
    "\trider STRING,\n" +
    "\tdriver STRING,\n" +
    "\tbegin_lat DOUBLE,\n" +
    "\tbegin_lon DOUBLE,\n" +
    "\tend_lat DOUBLE,\n" +
```

```
    "\tend_lon DOUBLE,\n" +
    "\tfare DOUBLE,\n" +
    "\tpartitionpath STRING,\n" +
    "\tts BIGINT\n" +
    ") WITH (\n" +
    "  'connector' = 'kafka',\n" +
    "  'topic' = 'hudi_source',\n" +
    "  'properties.bootstrap.servers' = '<kafka_server>:9092',\n" +
    "  'properties.group.id' = 'test-group-10001',\n" +
    "  'scan.startup.mode' = 'latest-offset',\n" +
    "  'format' = 'json'\n" +
    ")");
```

```
// Specify the Hudi target table
tableEnv.executeSql("CREATE TABLE hudi_cow (\n" +
    "uuid STRING PRIMARY KEY NOT ENFORCED,\n" +
    "rider STRING,\n" +
    "driver STRING,\n" +
    "begin_lat DOUBLE,\n" +
    "begin_lon DOUBLE,\n" +
    "end_lat DOUBLE,\n" +
    "end_lon DOUBLE,\n" +
    "fare DOUBLE,\n" +
    "partitionpath STRING,\n" +
    "ts BIGINT\n" +
    ") " +
    "partitioned by(partitionpath) " +
    "WITH (\n" +
    "  'connector' = 'hudi',\n" +
    "  'path' = 'cosn://<cos_bucket>/flink_hudi/hudi_cow',\n" +
    "  'fs.cosn.impl' = 'org.apache.hadoop.fs.CosFileSystem',\n" +
    "  'fs.AbstractFileSystem.cosn.impl' = 'org.apache.hadoop.fs.CosN',\n" +
    "  'fs.cosn.bucket.region' = 'ap-chongqing',\n" +
    "  'fs.cosn.credentials.provider' = 'org.apache.hadoop.fs.auth.SimpleCredentialProvider',\n" +
    "  'fs.cosn.userinfo.secretId' = '<secretId>',\n" +
    "  'fs.cosn.userinfo.secretKey' = '<secretKey>',\n" +
    "  'table.type' = 'COPY_ON_WRITE',\n" +
    "  'write.operation' = 'upsert',\n" +
    "  'hoodie.datasource.write.recordkey.field' = 'uuid',\n" +
    "  'write.precombine.field' = 'ts',\n" +
    "  'write.tasks' = '1'\n" +
    ")");
```

```
// Writing into Hudi tables using Flink SQL
tableEnv.executeSql("insert into hudi_cow select
uuid,rider,driver,begin_lat,begin_lon,end_lat,end_lon,fare,partitionpath,ts from tbl_kafka");
```

## Relevant Configuration

- Common Write Configurations

| Category | Default value | Description |
| --- | --- | --- |
| hoodie.datasource.write.table.name | _ | Specify the name of the Hudi table to be written into |
| hoodie.datasource.write.table.type | COPY_ON_WRITE | Specify the Hudi table type. Once this table type is specified, subsequent modifications to this parameter are prohibited. The optional values are: COPY_ON_WRITE, MERGE_ON_READ. |
| hoodie.datasource.write.operation | upsert | The operation type specified for writing to the Hudi table is currently supported in the following modes: UPSERT, DELETE, INSERT, BULK_INSERT, INSERT_OVERWRITE, and INSERT_OVERWRITE_TABLE. |
| hoodie.datasource.write.recordkey.field | uuid | This is used to specify the primary key for Hudi. Hudi tables require a unique primary key. |
| hoodie.datasource.write.partitionpath.field | _ | This is used to specify the partition key. This value, in conjunction with hoodie.datasource.write.keygenerator.class, can meet different partition scenarios. |
| hoodie.datasource.write.hive_style_partitioning | false | This is used to specify whether the partitioning method is consistent with Hive. It is recommended to set this value to true. |
| hoodie.datasource.write.precombine.field | ts | This value is used to merge and deduplicate rows with the same key before writing. |

- Compaction Configuration
  Compaction is used to merge the Base and Log files of the MOR table. For Merge-On-Read tables, data is stored in columnar Parquet files and row-based Avro files. Updates are recorded to incremental files, and then synchronous/asynchronous compaction is performed to generate new versions of columnar files. Merge-On-Read tables can reduce data ingestion latency. It is recommended to use a synchronous method to generate compaction scheduling plans and an asynchronous method to execute these plans.

| Category | Default value | Description |
| --- | --- | --- |
| hoodie.compact.schedule.inline | false | It is recommended to set it to true for whether to generate a compact plan after each task completion. |
| hoodie.compact.inline | false | Whether to execute the compression operation inline after a transaction is completed. Enabling this does not necessarily trigger the index operation every time, as there is a policy judgment following. |
| hoodie.compact.inline.trigger.strategy | CompactionTriggerStrategy.NUM_COMMITS | Compression strategy parameters, which include NUM_COMMITS, |

| | | |
|---|---|---|
| | | TIME_ELAPSED, NUM_AND_TIME, NUM_OR_TIME.<br>NUM_COMMITS determines whether to compress based on the number of commits.<br>TIME_ELAPSED determines whether to compress based on time, while NUM_AND_TIME determines whether to compress based on both the number of submissions and time.<br>NUM_OR_TIME determines whether to compress based on the number of submissions or time. |
| hoodie.compact.inline.max.delta.commits | 5 | Set the number of submissions after which the compression policy is triggered. Effective in NUM_COMMITS, NUM_AND_TIME, and NUM_OR_TIME policies. |
| hoodie.compact.inline.max.delta.seconds | 60 * 60 (1 Hour) | Set the duration after which the compression policy is triggered. This is effective in TIME_ELAPSED, NUM_AND_TIME, and NUM_OR_TIME policies. |
| hoodie.parquet.small.file.limit | 104857600(100MB) | Files smaller than this value are considered small files, and newly added data will be written into these small files as a priority. |

## Table-level Concurrency Control for Writing

If there is only one client writing to the Hudi table at the time of writing, there will be no data conflict. However, in practical applications, if multiple clients are writing at the same time, such as multiple stream programs needing to write to the same Hudi table simultaneously, write conflicts causing task failure may occur. We refer to this situation as concurrent writing. To solve the concurrent writing problem, DLC Metastore can be used to implement optimistic lock-based concurrent writing.

- Enable Concurrent Write Mechanism:

```
hoodie.write.concurrency.mode=optimistic_concurrency_control

hoodie.cleaner.policy.failed.writes=LAZY
```

- Set the concurrency lock mode to DLC Metastore mode:

```
hoodie.write.lock.provider=org.apache.hudi.hive.HiveMetastoreBasedLockProvider

hoodie.write.lock.hivemetastore.database=<database_name>

hoodie.write.lock.hivemetastore.table=<table_name>
```

- Example of DLC Spark MultiWriter:

```
kafkaDF.writeStream
  .option("checkpointLocation","cosn://<cos_bucket>/spark_hudi/spark_ck/writer2")
  .trigger(Trigger.ProcessingTime(10, TimeUnit.SECONDS))
  .queryName("write hudi")
  .foreachBatch((batchDF:DataFrame,_:Long)=>{
   batchDF.write
     .mode(SaveMode.Append)
     .format("hudi")
     .option("hoodie.datasource.write.table.type","MERGE_ON_READ")
     .option("hoodie.datasource.write.precombine.field","ts")
     .option("hoodie.datasource.write.recordkey.field","uuid")
     .option("hoodie.datasource.write.partitionpath.field","partitionpath")
     .option("hoodie.datasource.write.table.name","multi_writer")
     .option("hoodie.write.concurrency.mode","optimistic_concurrency_control")
     .option("hoodie.cleaner.policy.failed.writes","LAZY")

.option("hoodie.write.lock.provider","org.apache.hudi.hive.transaction.lock.HiveMetastoreBasedLockPr
ovider")
     .option("hoodie.write.lock.hivemetastore.database","spark_hudi")
     .option("hoodie.write.lock.hivemetastore.table","multi_writer")
     .save("cosn://<cos_bucket>/spark_hudi/multi_writer")
  }).start().awaitTermination()
```

# Hudi Data Query

Last updated: 2024-01-10 16:43:09

The query operations on Hudi tables are performed over the three types of Hudi views. You can choose the appropriate view for querying based on your specific requirements.

## Real-time Snapshot View (Snapshot Queries)

This view allows querying of the latest COMMIT snapshot data. For Merge On Read tables, it requires online merging of base data in column storage and real-time data in logs during the query. For Copy On Write tables, it allows querying of the latest version of Parquet data. Both Copy On Write and Merge On Read tables support this type of query.

- SparkSQL Sample:

```
SELECT count(*) FROM DataLakeCatalog.hudi_spark.hudi_test
```

- Spark API Sample:

```
spark.read.
  format("hudi")
  .option(QUERY_TYPE_OPT_KEY,QUERY_TYPE_SNAPSHOT_OPT_VAL)
  .option(BEGIN_INSTANTTIME_OPT_KEY,"20221009003522620")
  .load("cosn://<cos_bucket>/spark_hudi/hudi_cow_sync")
  .createOrReplaceTempView("hudi_test")
.spark.sql("select count(*) from hudi_test").show()
```

## Incremental View (Incremental Queries)

This view only queries the files of the newly written data set. It requires specifying an instant time of a Commit/Compaction (an Instant on the Timeline) as a condition to query the new data after this condition. Both Copy On Write and Merge On Read tables support this type of query.

- SparkSQL Sample:
  In the advanced settings of the sparksql engine, add the following two parameters:

```
hoodie.datasource.query.type=incremental
hoodie.datasource.read.begin.instanttime=20221009003522620
SELECT count(*) FROM DataLakeCatalog.hudi_spark.hudi_test
```

- Spark API Sample:

```
spark.read.
 format("hudi")
 .option(QUERY_TYPE_OPT_KEY,QUERY_TYPE_INCREMENTAL_OPT_VAL)
 .option(BEGIN_INSTANTTIME_OPT_KEY,"20221009003522620")
 .load("cosn://<cos_bucket>/spark_hudi/hudi_cow_sync")
 .show(false)
```

## Read Optimized View (Read Optimized Queries)

This view exposes only the base/column files (Parquet) in the latest file slice to the query, ensuring the same columnar query performance as non-Hudi columnar datasets. This view is an optimization of snapshot queries for Merge On Read table types, reducing query latency caused by online merging of log data at the expense of query data timeliness.

- SparkSQL Sample:
  In the advanced settings of the sparksql engine, add the following two parameters:
  hoodie.datasource.query.type=read_optimized
  hoodie.datasource.read.begin.instanttime=20221009003522620

```
SELECT count(*) FROM  DataLakeCatalog.hudi_spark.hudi_test
```

- Spark API Sample:

```
spark.read.
 format("hudi")
 .option(QUERY_TYPE_OPT_KEY,QUERY_TYPE_READ_OPTIMIZED_OPT_VAL)
 .load("cosn://<cos_bucket>/spark_hudi/hudi_cow_sync")
 .show(false)
```

# System Restraints
# Metadata Information

Last updated: 2024-01-10 16:43:16

The following lists the limits on the numbers of databases, data tables, attribute columns, and partitions.

| Item | Maximum Quantity |
| --- | --- |
| Number of Databases per Account | 1,000 |
| Number of Data Tables per Account | 10,000 |
| Number of Tables per Database | 4,096 |
| Number of Columns per Data Table | 4,096 |
| Number of Partitions per Table | 10,000 |
| Number of Partitions per Master Account | 1,000,000 |
| Maximum Number of Fields per Table | 4096 |
| Number of Custom Functions per Account | 100 |
| Number of Catalogs that can be Created | 20 |

## Databases

- Name: Should not exceed 127 characters, and within the same data link, identical database names are not permitted.
- Description: Should not exceed 2048 characters.
- External Table Data Address (COS Address): 888 characters (limited by the length of the COS path).
- Parameters: In the form of `Map<string:string>`, each parameter is limited to 127 characters, with a total length limit of 3000 characters.

## Data Table/View

- Name: Should not exceed 127 characters, and within the same database, identical table names are not permitted.
- Description: Should not exceed 1000 characters.
- External Table Data Address (COS Address): Should not exceed 888 characters (limited by the length of the COS path).
- Parameters: In the form of `Map<string:string>`, each parameter can contain up to 127 characters, with a total character limit of 512,000.

## Attribute Column

- Name: Should not exceed 127 characters, and within the same table, identical attribute column names are not permitted.
- Description: Should not exceed 256 characters.

- Type: Should not exceed 131,072 characters. Exceeding this limit will prevent creation.

## Partition

- Partition Field Name: Should not exceed 127 characters.

# Computing Task

Last updated: 2024-01-10 16:44:26

The size limit for a single SQL statement is 2 MB.