数据湖计算 DLC

开发指南

腾讯云

文档目录

开发指南
使用 SDK 数据写入

最近更新时间：2024-11-28 10:44:42

概述

数据湖计算 DLC 提供用户通过 Java SDK 数据导入。

应用场景

通过 Java sdk 将源数据库的增量变更同步到 DLC 原生表，完成源数据入湖。

适用场景：

1. 需要实时处理数据流的业务场景。
2. 熟悉 Java 并需要自定义逻辑处理的开发人员。

前置条件

- 正确开通 DLC，已完成用户权限配置，开通托管存储。
- 正确创建 DLC 数据库。
- 正确配置 DLC 数据库数据优化，详细配置请参见 开启数据优化 。

操作步骤

步骤1：下载依赖 Jar

请手动下载依赖 SDK jar 包 dlc-bridage.jar 。

步骤2：编写 Java 代码

```
package org.example;
// 建表 create table ingest_stream(id int,name string);

import com.gotocompany.depot.dlc.models.SinkRecord;
import com.tencent.dlc.DlcClient;
import com.tencent.dlc.RealtimeStream;
import com.tencent.dlc.RowStream;

public class RealtimeStreamDemo {
    public static void main(String[] args) throws Exception {

        String endpoint = "dlc.tencentcloudapi.com";
```

```
        String secretId ="";
        String secretKey ="";
        String database ="";
        String region = "";
        String table = "";
        DlcClient client = DlcClient.newBuilder()
                .endpoint(endpoint)
                .secretId(secretId)
                .secretKey(secretKey)
                .region(region)
                .build();

        RowStream stream = client.newRealtimeStreamBuilder()
                .operate(RowStream.RealTimeOperate.APPEND_ONLY)
                .database(database)
                .table(table)
                .commitInterval(5000)
                .build();

        for (int t = 0; t < 1000; t++) {
            SinkRecord row = stream.createRow();
            row.setRow("id", t);
            row.setRow("name", String.valueOf(t));
            stream.apply(row);
        }
        // 只有调用 flush 之后数据才可见
        ((RealtimeStream) stream).flush();

        // 必须调用 stream close接口，close 时会隐含执行 flush
        stream.close();
        client.close();
    }
}
```

```
<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>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

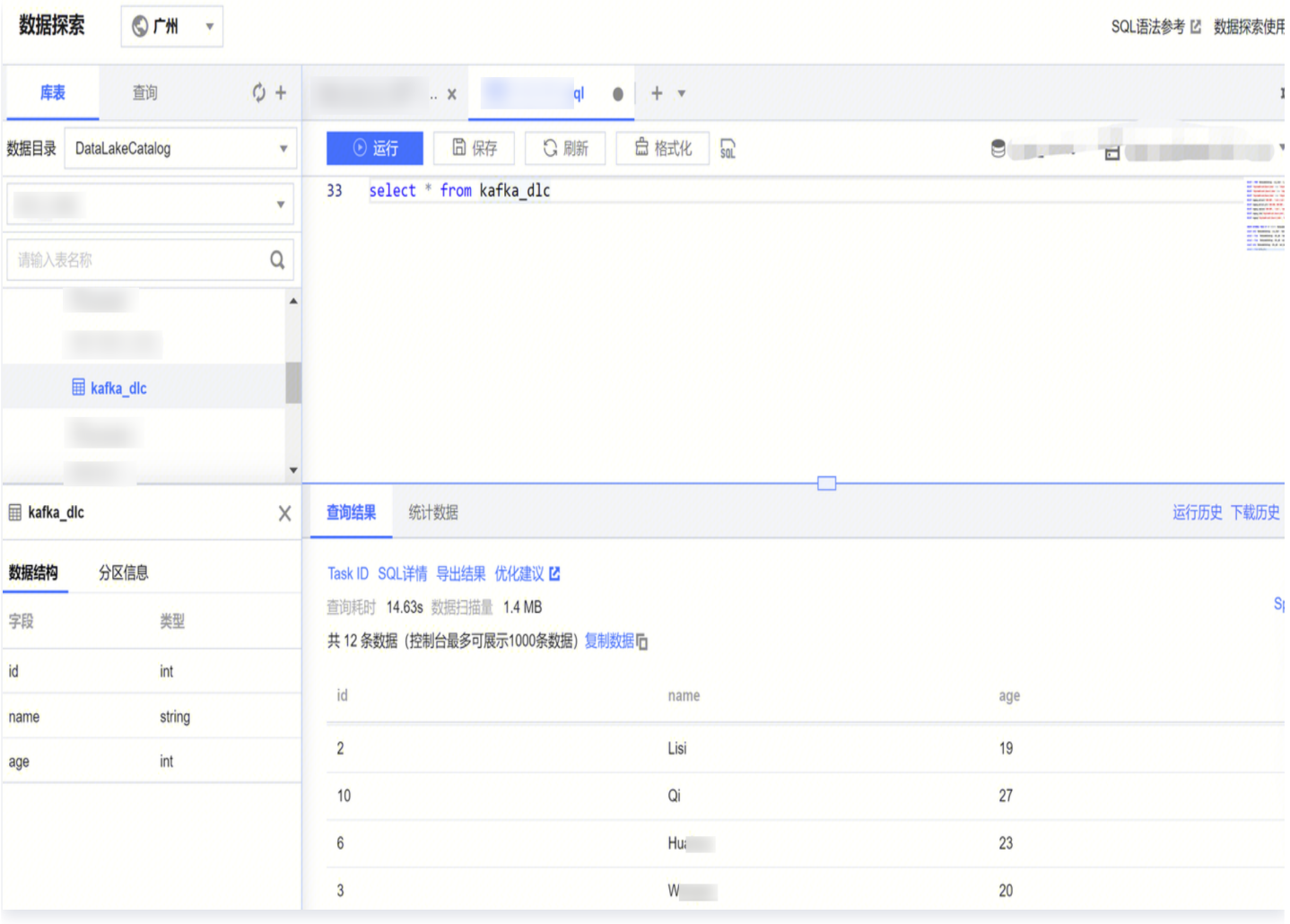
```
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>


    <dependency>
      <groupId>com.tencent.cloud.dlc</groupId>
      <artifactId>dlc-bridage</artifactId>
      <version>1.0</version>
    </dependency>

  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>8</source>
          <target>8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

步骤3：DLC 新建目标表

新建目标表详情可参见 DLC 原生表操作配置 。

步骤4：执行 SDK 程序

登录 DLC 控制台，单击**数据探索**，查询目标表数据。


数据探索
广州
SQL语法参考
数据探索使用
库表
查询
数据目录
DataLakeCatalog
运行
保存
刷新
格式化
SQL
33 select * from kafka_dlc
请输入表名称
kafka_dlc
kafka_dlc
数据结构
分区信息
字段
类型
id
int
name
string
age
int
查询结果
统计数据
运行历史
下载历史
Task ID
SQL详情
导出结果
优化建议
查询耗时 14.63s
数据扫描量 1.4 MB
共 12 条数据（控制台最多可展示1000条数据）
复制数据
id
name
age
2
Lisi
19
10
Qi
27
6
23
3
20

SparkJar 作业开发指南

最近更新时间：2024-07-03 14:15:21

应用场景

DLC 完全兼容开源 Apache Spark，支持用户编写业务程序在 DLC 平台上对数据进行读写和分析。本示例演示通过编写 Java 代码在 COS 上读写数据和在 DLC 上建库表、读写表的详细操作，帮助用户在 DLC 上完成作业开发。

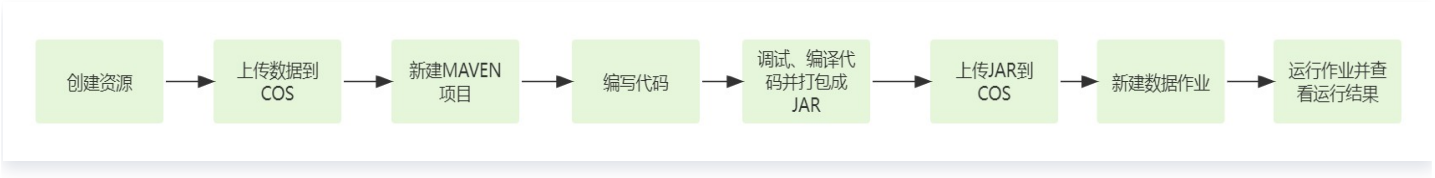
环境准备

依赖： JDK1.8 Maven IntelliJ IDEA

开发流程

开发流程图

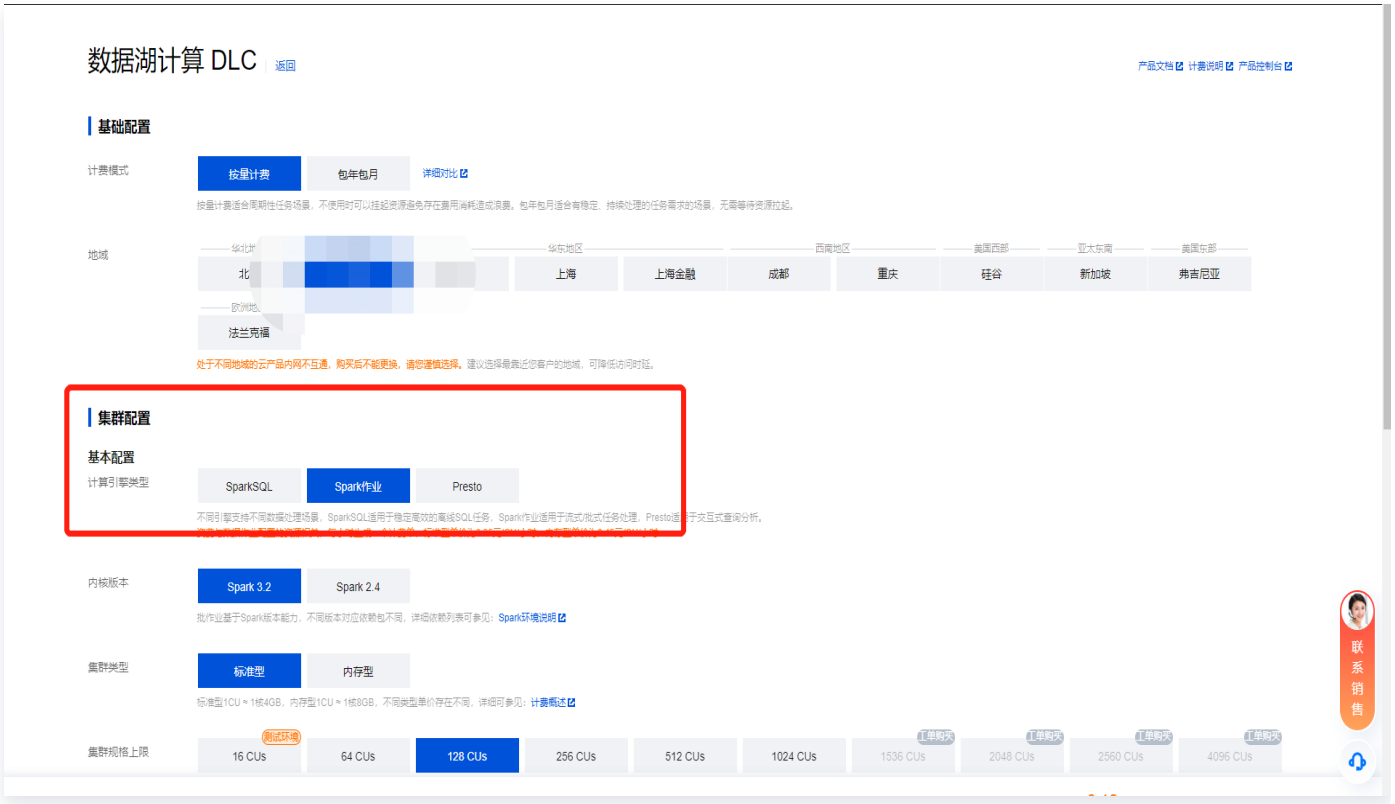
DLC Spark JAR 作业开发流程图如下：



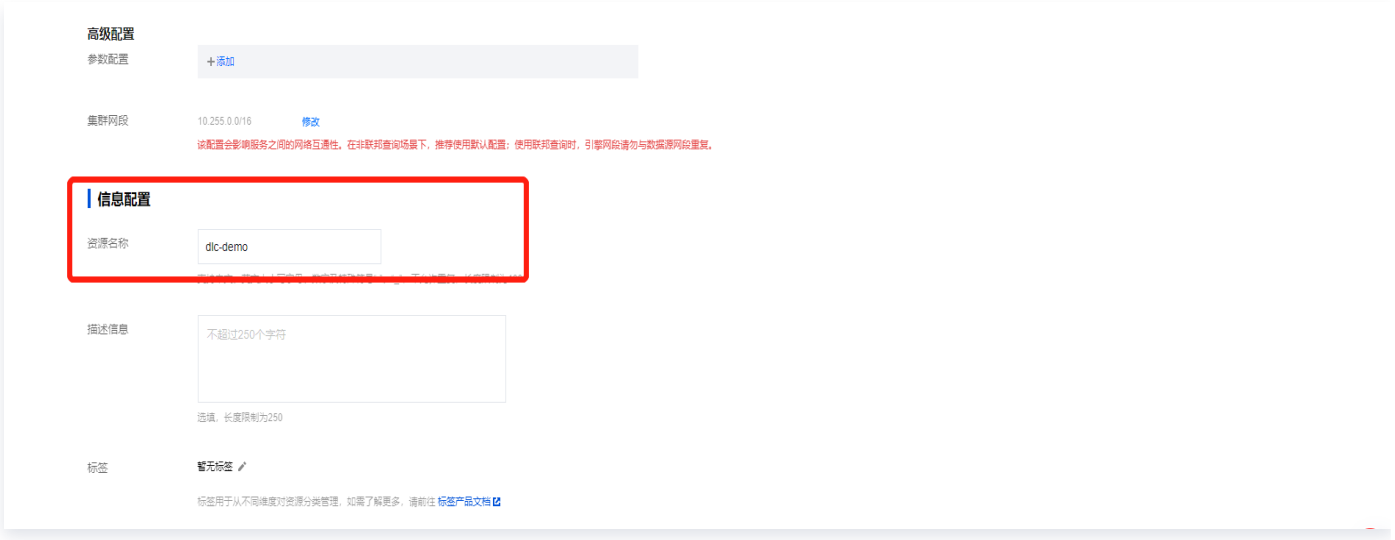
创建资源

第一次在 DLC 上运行作业，需新建 Spark 作业计算资源，例如新建名称为 "dlc-demo" 的 Spark 作业资源。

1. 登录 数据湖计算 DLC 控制台，选择服务所在区域，在导航菜单中单击**数据引擎**。
2. 单击左上角**创建资源**，进入资源配置购买页面。
3. 在**集群配置 > 计算引擎类型**项选择 Spark 作业引擎。



信息配置 > 资源名称 项填写 “dlc-demo”。新建资源详细介绍请参见 购买独享数据引擎 。



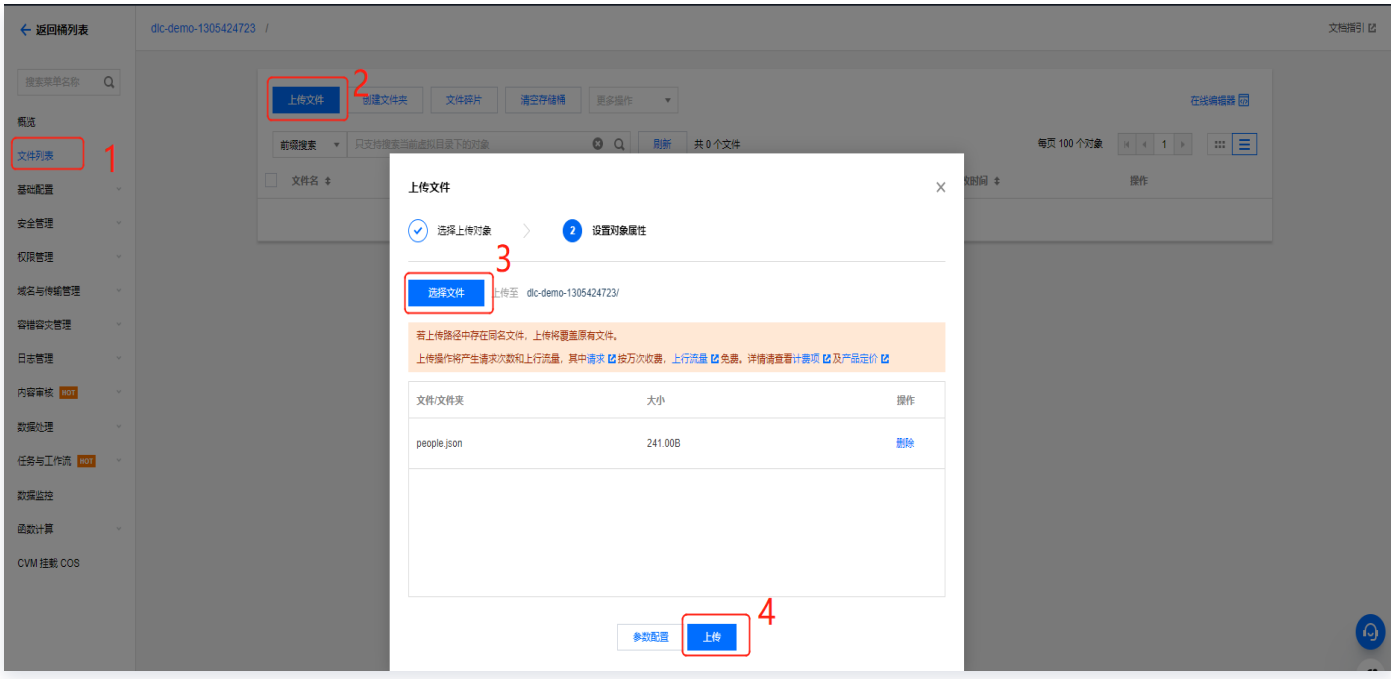
4. 单击**立即开通**，确认资源配置信息。
5. 确认信息无误后，单击**提交**，完成资源配置。

上传数据到 COS

创建名称为 “dlc-demo”的存储桶，上传 people.json 文件，用作从 COS 读写数据的示例，people.json 文件的内容如下：

```
{"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. 登录 对象存储 COS 控制台，在左侧菜单导航中单击 **存储桶列表**。
2. 创建存储桶：单击左上角**创建存储桶**，名称项填写 “dlc-dmo”，单击**下一步**完成配置。
3. 上传文件：单击**文件列表 > 上传文件**，选择本地“people.json”文件上传到“dlc-demo-1305424723”桶里（-1305424723是建桶时平台生成的随机串），单击**上传**，完成文件上传。新建存储桶详情可参见 创建存储桶 。



新建 Maven 项目

1. 通过 IntelliJ IDEA 新建一个名称为“demo”的 Maven 项目。
2. 添加依赖：在 pom.xml 中添加如下依赖：

```
<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>
```

编写代码

编写代码功能为从 COS 上读写数据和在 DLC 上建库、建表、查询数据和写入数据。

1. 从 COS 上读写数据代码示例：

```
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.创建SparkSession
        SparkSession spark = SparkSession
                .builder()
                .appName("Operate data on cos")
                .config("spark.some.config.option", "some-value")
                .getOrCreate();
        //2.读取cos上的json文件生成数据集,支持多种类型的文件，如
json,csv,parquet,orc,text
        String readPath = "cosn://dlc-demo-1305424723/people.json";
        Dataset<Row> readData = spark.read().json(readPath);
        //3.对数据集做业务计算操作生成结果数据，计算支持API和SQL形式，这里生成临
时表用sql读数据
```

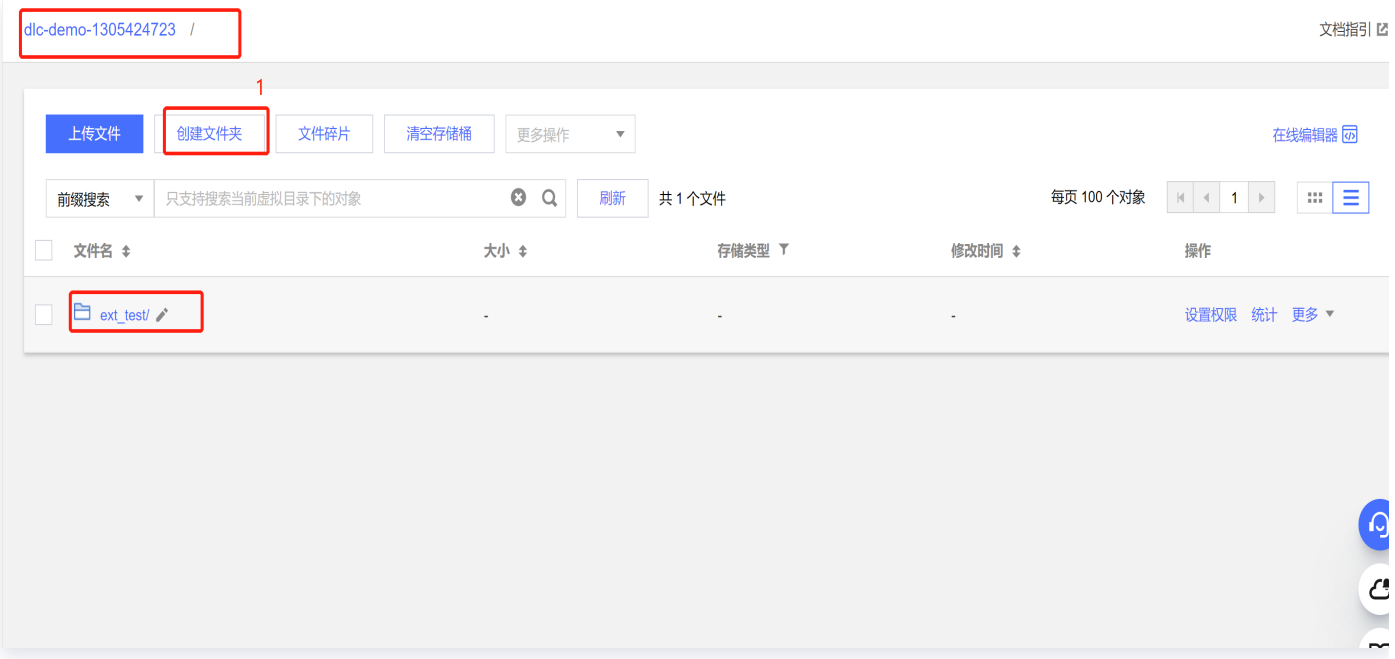
```
        readData.createOrReplaceTempView("people");
        Dataset<Row> result = spark.sql("SELECT * FROM people where
age > 3");
        //4.结果数据保存到cos
         String writePath = "cosn://dlc-demo-
1305424723/people_output";
        //写入支持多种类型的文件，如 json,csv,parquet,orc,text
        result.write().mode(SaveMode.Append).json(writePath);
        spark.read().json(writePath).show();
        //5.关闭session
        spark.stop();
    }
}
```

2. DLC 上建库、建表、查询数据和写入数据：

```
package com.tencent.dlc;

import org.apache.spark.sql.SparkSession;


public class DbService {

    public static void main(String[] args) {
        //1.初始化SparkSession
        SparkSession spark = SparkSession
                .builder()
                .appName("Operate DB Example")
                .getOrCreate();
        //2.建数据库
        String dbName = " `DataLakeCatalog`.`dlc_db_test` ";
        String dbSql = "CREATE DATABASE IF NOT EXISTS" + dbName + "
COMMENT 'demo test'";
        spark.sql(dbSql);
        //3.建内表
        String tableName = "`test`";
        String tableSql = "CREATE TABLE IF NOT EXISTS " + dbName + "."
+ tableName
                + "(`id` int,`name` string, `age` int)";
        spark.sql(tableSql);
        //4.写数据
        spark.sql("INSERT INTO " + dbName + "." + tableName + "VALUES
(1,'Andy',12),(2,'Justin',3) ");
```

```
        //5.查询数据
        spark.sql(" SELECT * FROM " + dbName + "." +
tableName).show();
        //6.建外表
        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.写外表数据
        spark.sql("INSERT INTO " + dbName + "." + extTableName +
"VALUES (1,'LiLy',12),(2,'Lucy',3) ");
        //8.查询外表数据
        spark.sql(" SELECT * FROM " + dbName + "." +
extTableName).show();
        //9.关闭Session
        spark.stop();
    }
}
```

建外表时，需按照 上传数据到COS的步骤 先在桶里建对应表名文件夹保存表文件



调试、编译代码并打成 JAR 包

通过 IntelliJ IDEA 对 demo 项目编译打包，在项目 target 文件夹下生成JAR包 demo-1.0-SNAPSHOT.jar。

上传 JAR 包到 COS

登录 COS 控制台，参考 **上传数据到 COS** 的步骤将 demo-1.0-SNAPSHOT.jar 上传到 COS。

新建 Spark Jar 数据作业

创建数据作业前，您需先完成数据访问策略配置，保证数据作业能安全地访问到数据。配置数据访问策略详情请参见 配置数据访问策略 。如已配置数据策略名称为：qcs::cam::uin/100018379117:roleName/dlc-demo。

1. 登录 数据湖计算 DLC 控制台，选择服务所在区域，在导航菜单中单击**数据作业**。
2. 单击左上角**创建作业**，进入创建页面。
3. 在作业配置页面，配置作业运行参数，具体说明如下：

配置参数	说明
作业名称	自定义 Spark Jar 作业名称，例如：cosn-demo
作业类型	选择 **批处理类型**
数据引擎	选择 **创建资源** 步骤创建的 dlc-demo 计算引擎
程序包	选择 COS，在 **上传 JAR 包到 COS** 步骤上传的 JAR 包 demo-1.0-SNAPSHOT.jar
主类（Main Class）	根据程序代码填写，如： • 从 COS 上读写数据填： com.tencent.dlc.CosService • 在 DLC 上建库、建表等填：com.tencent.dlc.DbService
数据访问策略	选择该步骤前创建的策略 qcs::cam::uin/100018379117:roleName/dlc-demo

其他参数值保持默认。

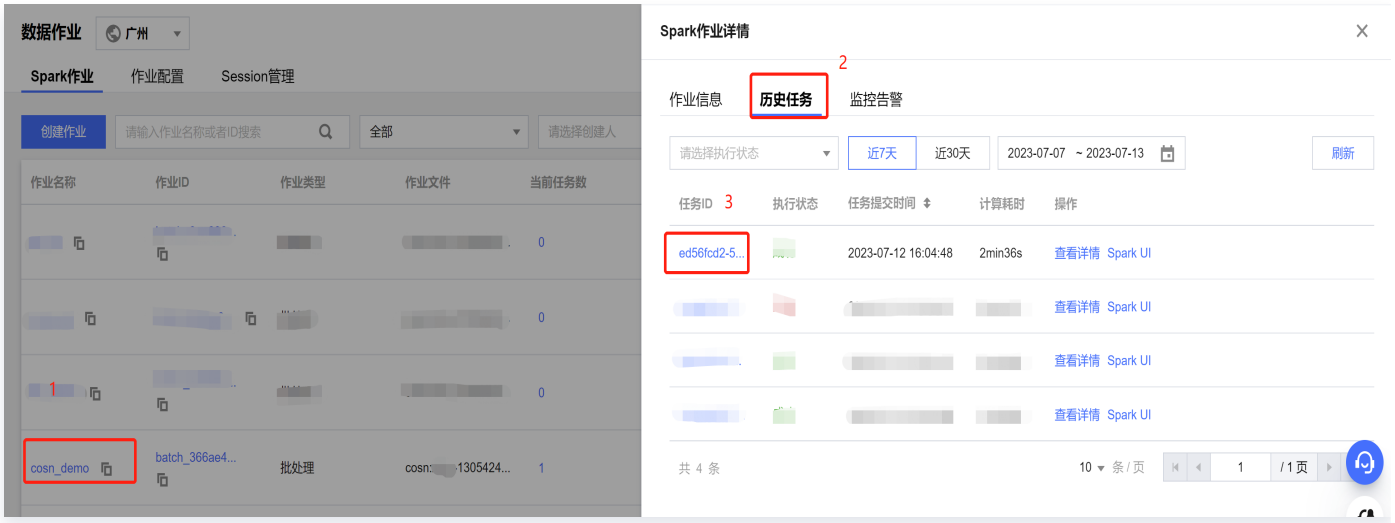
4. 单击**保存**，在**Spark 作业**页面可以看到创建的作业。

运行并查看作业结果

1. 运行作业：在**Spark 作业**页面，找到新建的作业，单击**运行**，即可运行作业。
2. 查看作业运行结果: 可查看作业运行日志和运行结果。

查看作业运行日志

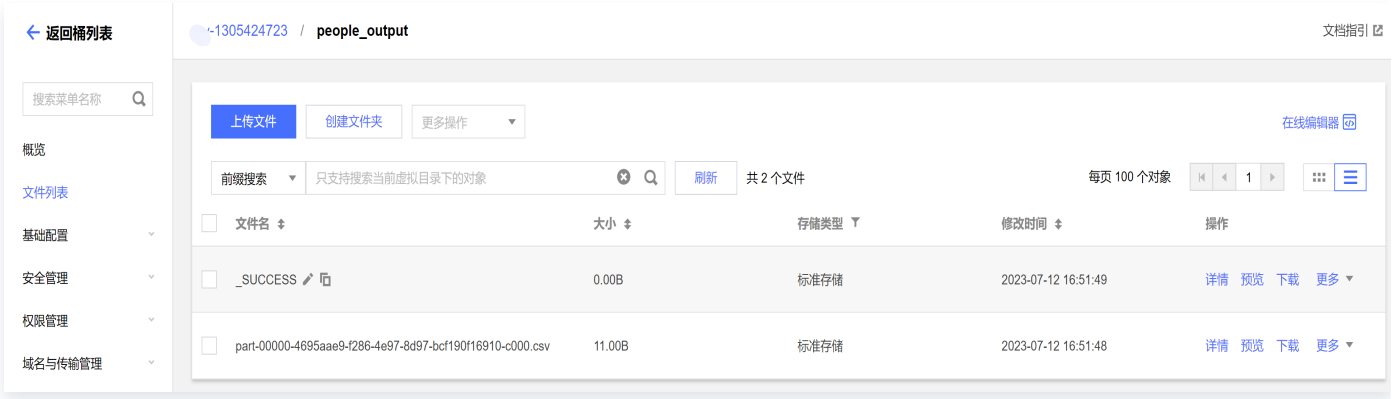
1. 单击**作业名称 > 历史任务**，查看任务运行状态。



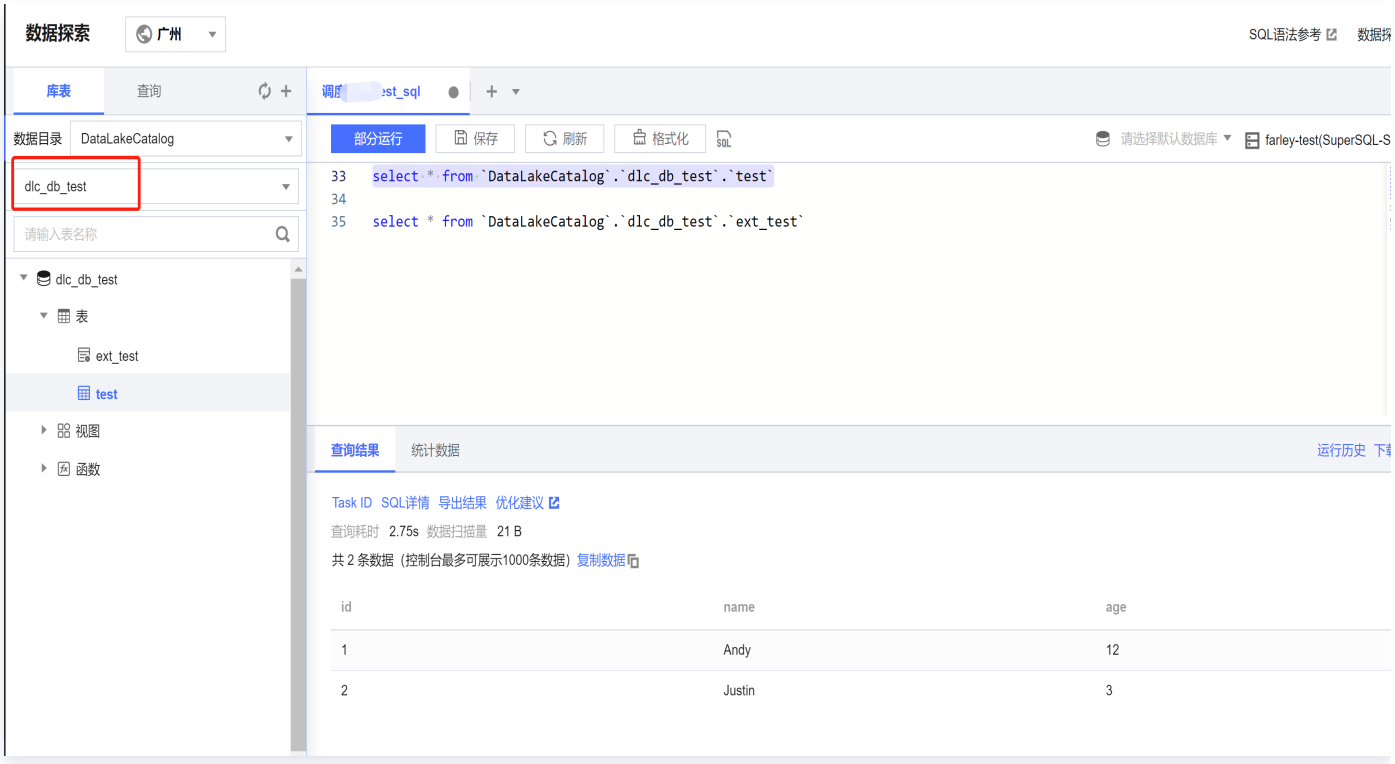
2. 单击**任务ID > 运行日志**，查看作业运行日志：

查看作业运行结果

1. 运行从 COS 读写数据示例，则到 COS 控制台 查看数据写入结果。



2. 运行在 DLC 上建表、建库，则到 DLC **数据探索**页面查看建库、建表。



PySpark 作业开发指南

最近更新时间：2025-01-20 12:17:52

应用场景

DLC 支持 Python 语言编写的程序运行作业 。本示例演示通过编写 Python 代码在 对象存储（COS）上读写数据和在 DLC 上建库表、读写表的详细操作，帮助用户在 DLC 上完成作业开发。

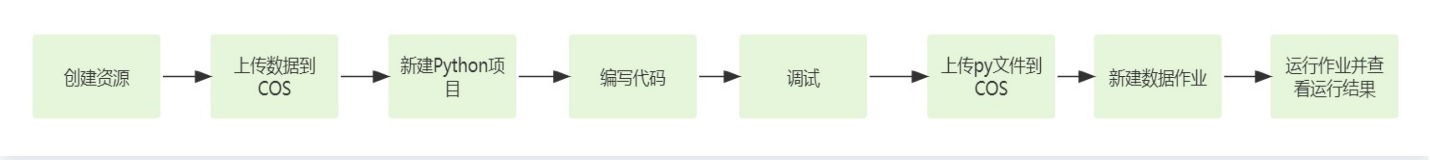
环境准备

依赖：PyCharm 或其他 Python 编程开发工具。

开发流程

开发流程图

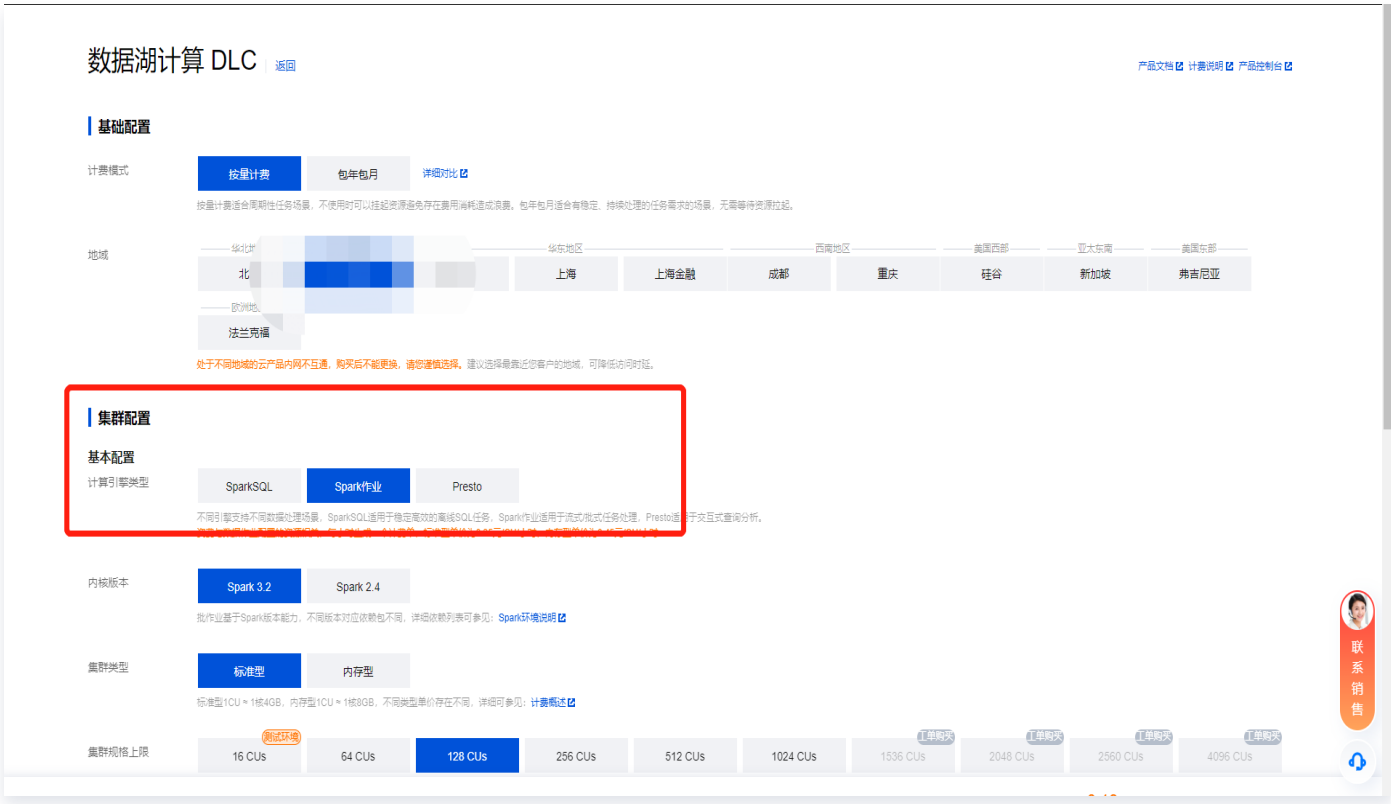
DLC Spark JAR 作业开发流程图如下：



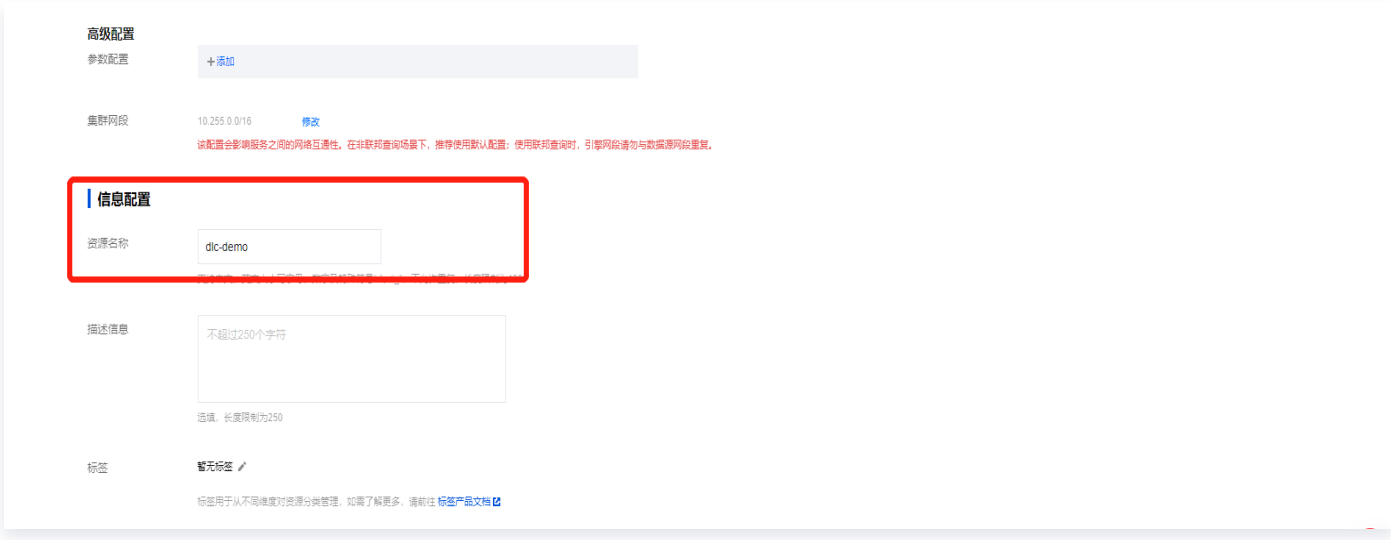
创建资源

第一次在 DLC 上运行作业，需新建 Spark 作业计算资源，例如新建名称为 "dlc-demo" 的 Spark 作业资源。

1. 登录 数据湖计算 DLC 控制台，选择服务所在区域，在导航菜单中单击**数据引擎**。
2. 单击左上角**创建资源**，进入资源配置购买页面。
3. 在**集群配置 > 计算引擎类型**选项选择 Spark 作业引擎。



信息配置 > 资源名称填写 “dlc-demo”。新建资源详细介绍请参见**购买独享数据引擎**。



4. 单击**立即开通**，确认资源配置信息。
5. 确认信息无误后，单击**提交**，完成资源配置。

上传数据到 COS

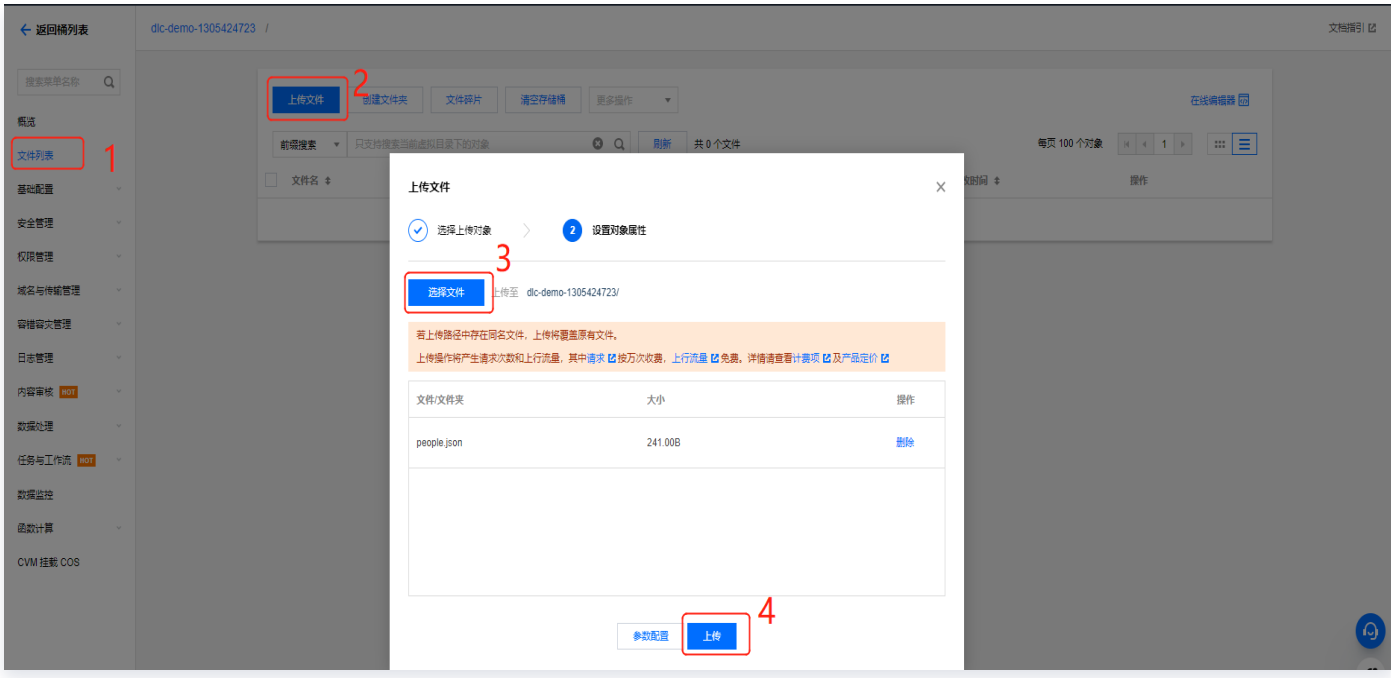
创建名称为 “dlc-demo”的存储桶，上传people.json文件，供 从COS 读写数据示例用，people.json 文件的内容如下：

```
{"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. 登录 对象存储 COS 控制台，在左侧菜单导航中单击 **存储桶列表**。
2. 创建存储桶：

   单击左上角 **创建存储桶**，名称项填写 “dlc-dmo”，单击**下一步**完成配置。
3. 上传文件：

   单击**文件列表 > 上传文件**，选择本地“people.json”文件上传到“dlc-demo-1305424723”桶里（-1305424723是建桶时平台生成的随机串），单击**上传**，完成文件上传。新建存储桶详情请参见 创建存储桶 。



新建 Python 项目

通过 PyCharm 新建一个名称为“demo”的 项目。

编写代码

1. 新建 cos.py 文件，编写代码，功能为从 COS 上读写数据和在 DLC 上建库、建表、查询数据和写入数据。

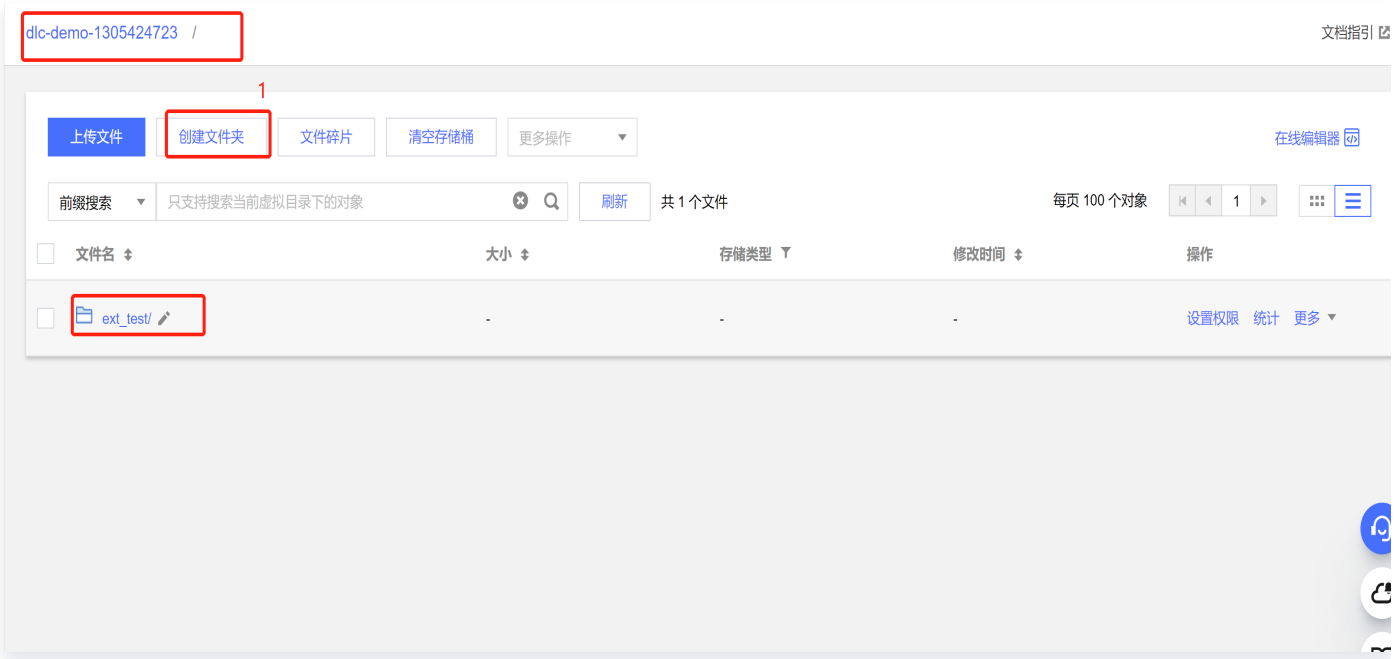
```
import sys
from pyspark.sql import SparkSession
from pyspark.sql import Row

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

    # 1.读cos上的数据 支持多种类型的文件 如 json,csv,parquet,orc,text
    read_path = "cosn://dlc-demo-1305424723/people.json"
    peopleDF = spark.read.json(read_path)

    # 2.对数据做操作
    peopleDF.createOrReplaceTempView("people")
    data_src = spark.sql("SELECT * FROM people WHERE age BETWEEN 13
AND 19")
    data_src.show()

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

    spark.stop()
```

2. 新建 db.py 文件，编写代码，功能为 DLC 上建库、建表、查询数据和写入数据。

```
from os.path import abspath

from pyspark.sql import SparkSession



if __name__ == "__main__":

    spark = SparkSession \
```

```
        .builder \
        .appName("Operate DB Example") \
        .enableHiveSupport()\
        .getOrCreate()

    # 1.建数据库
    spark.sql("CREATE DATABASE IF NOT EXISTS
`DataLakeCatalog`.`dlc_db_test_py` COMMENT 'demo test' ")
    # 2.建内表
    spark.sql("CREATE TABLE IF NOT EXISTS
`DataLakeCatalog`.`dlc_db_test_py`.`test`(`id` int,`name` string,`age`
int) ")
    # 3.写内数据
    spark.sql("INSERT INTO `DataLakeCatalog`.`dlc_db_test_py`.`test`
VALUES (1,'Andy',12),(2,'Justin',3) ")
    # 4.查内数据
    spark.sql("SELECT * FROM `DataLakeCatalog`.`dlc_db_test_py`.`test`
").show()

    # 5.建外表
    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.写外数据
    spark.sql("INSERT INTO
`DataLakeCatalog`.`dlc_db_test_py`.`ext_test` VALUES (1,'Andy',12),
(2,'Justin',3) ")
    # 7.查外数据
    spark.sql("SELECT * FROM
`DataLakeCatalog`.`dlc_db_test_py`.`ext_test` ").show()
    spark.stop()
```

建外表时，可按照 上传数据到 COS 的步骤 先在桶里建对应表名文件夹保存表文件。



调式

PyCharm 调式无语法错误。

上传 py 文件到 COS

登录 COS 控制台，参考上文**上传数据到 COS** 的步骤将 cos.py、db.py上传到 COS。

新建 Spark Jar 数据作业

创建数据作业前，您需先完成数据访问策略配置，保证数据作业能安全地访问到数据。配置数据访问策略详情请参见 配置数据访问策略 。如已配置数据策略名称为：qcs::cam::uin/100018379117:roleName/dlc-demo。

1. 登录 数据湖计算 DLC 控制台，选择服务所在区域，在导航菜单中单击**数据作业**。
2. 单击左上角**创建作业**按钮，进入创建页面。
3. 在作业配置页面，配置作业运行参数，具体说明如下：

配置参数	说明
作业名称	自定义 Spark 作业名称，例如：cosn_py
作业类型	选择 **批处理类型**
数据引擎	选择 **创建资源** 步骤创建的 dlc-demo 计算引擎

程序包	选择 COS，在 **上传 py 文件到 COS** 步骤的上传 py 文件： • 从 COS 上读写数据就选择： cosn://dlc-demo-1305424723/cos.py • 在 DLC 上建库、建表等选择：cosn://dlc-demo-1305424723/db.py
数据访问策略	选择该步骤前创建的策略 qcs::cam::uin/100018379117:roleName/dlc-demo

其他参数值保持默认。

4. 单击**保存**，在 **Spark 作业**页面可以看到创建的作业。

运行并查看作业结果

1. 运行作业：在**Spark 作业**页面，找到新建的作业，单击**运行**，即可运行作业。
2. 查看作业运行结果：可查看作业运行日志和运行结果。

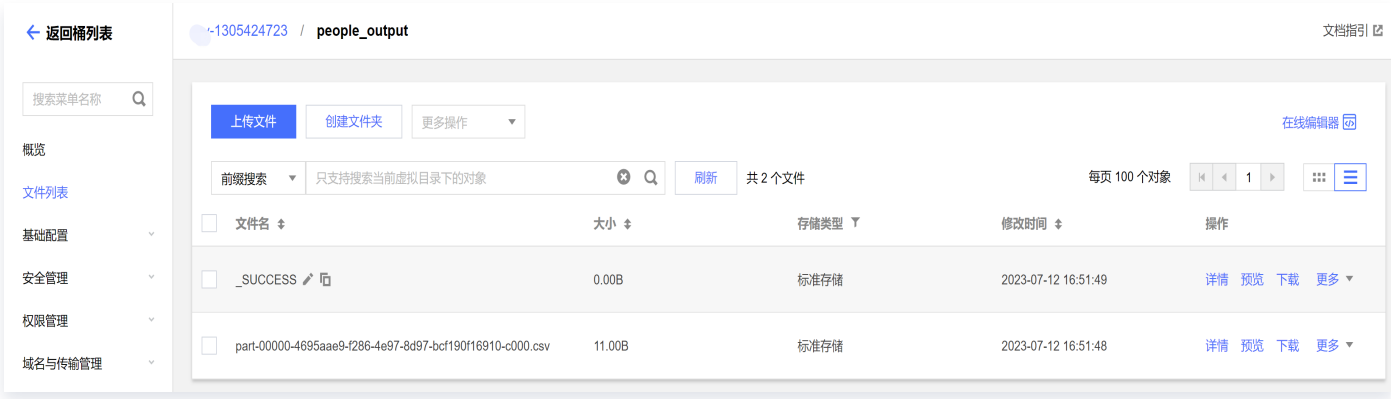
查看作业运行日志

1. 单击**作业名称 > 历史任务** 查看任务运行状态：

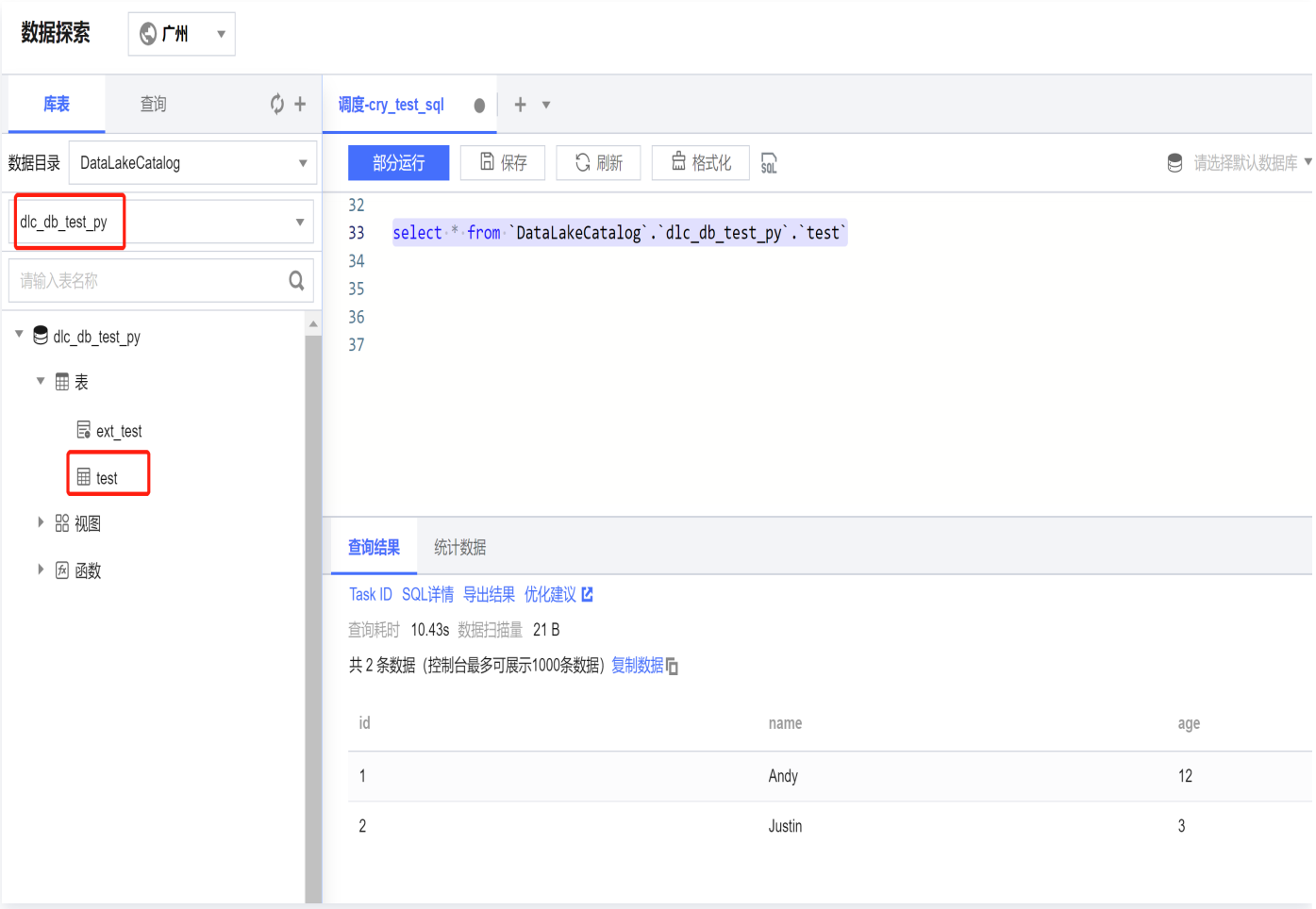
2. 单击**任务ID > 运行日志**，查看作业运行日志。

查看作业运行结果

1. 运行从 COS 读写数据示例，则到 COS 控制台查看数据写入结果。



2. 运行在 DLC 上建表、建库，则到 DLC 数据探索页面查看建库、建表。



查询性能优化指南

最近更新时间：2025-04-21 17:21:22

前言

为了提升任务执行效率，您可以首先对现有任务或引擎进行洞察分析（仅 Spark）查看是否有可调优空间，其次 DLC 在计算过程中有许多优化措施，例如数据治理、Iceberg 索引、缓存等，您可以结合这些优化措施对任务做更全面的优化。正确使用不仅可以减少不必要的扫描费用，甚至可以提升几倍甚至几十倍的效率。
下面提供一些不同层面的优化思路。

Spark

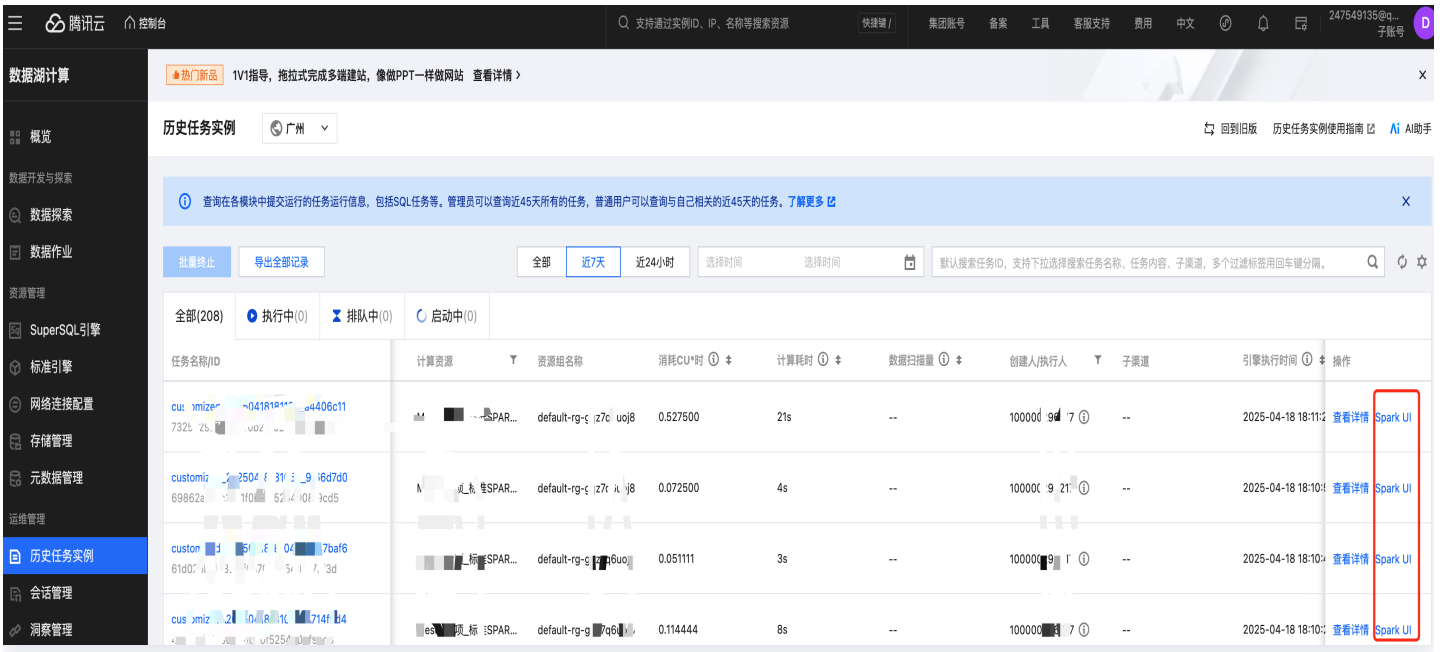
任务洞察

基于 Spark 引擎采集的各种 Metrics 信息，任务洞察模块实现了算法自动分析 Metrics 指标，为您提供任务的洞察情况，建议先基于推荐的参数和调优方案实施。
可以解决常见的 SQL 问题，包括：

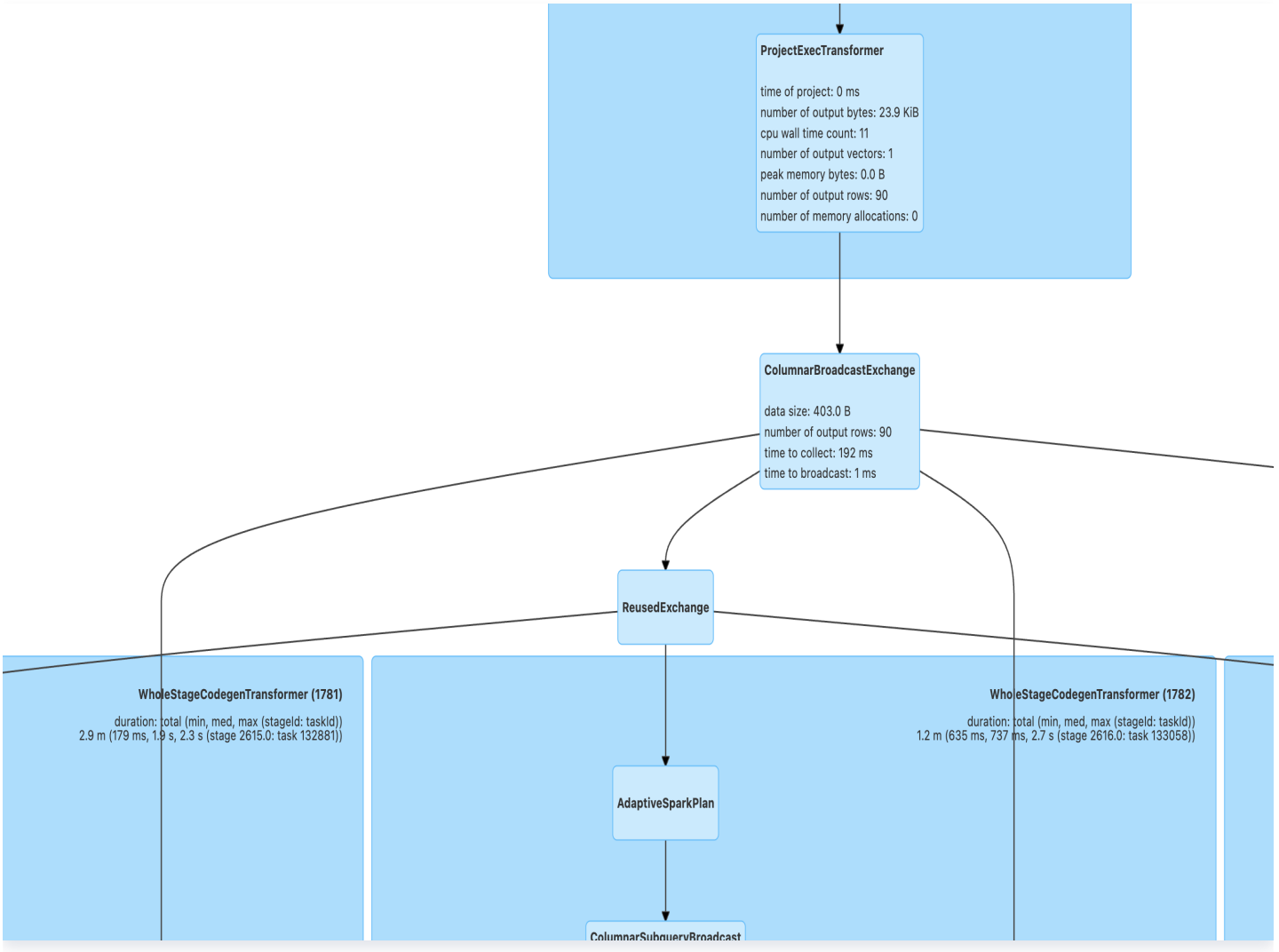
- 资源抢占
- 数据倾斜
- 磁盘不足或内存 OOM
- 慢 Task
- 小文件过多
- Shuffle 不合理

执行计划分析

当任务洞察仍然无法满足性能调优时，根据Spark UI提供的当前任务详细的执行计划，可以进一步分析 SQL 的瓶颈。



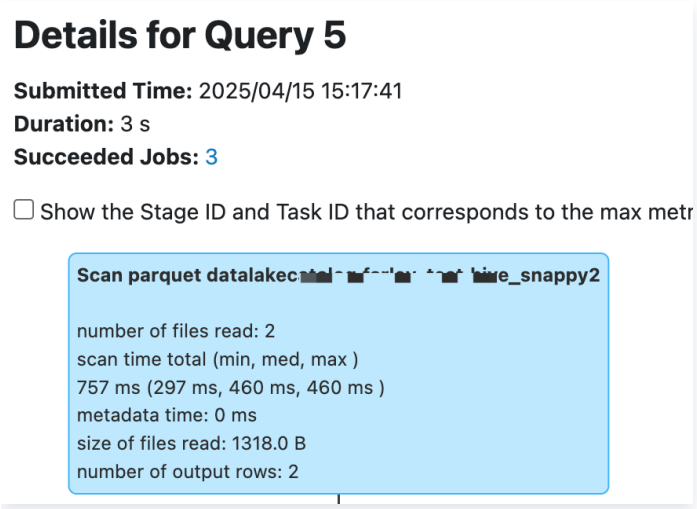
Spark UI 可以后从 SQL DataFrame 页面查看执行计划。



执行计划的查看和分析可以结合 Stages 页面的 Stage 耗时，按照以下步骤及优化思路。优化前，建议提前对输入、Join、输出的数据量级有初步的判断。

第一步：分析 Scan 算子

Scan 算子是负责读取所有库表的操作，Stages 页面包含有 Input 数据的 Stage 一般对应 Scan 算子，当这种 Stage 耗时长时需要重点查看 Scan 算子的指标。



Scan算子示例

以下 Metrics 信息在 Hive/Iceberg 表上名称有所不同，但是指标内容及含义基本相同。

- number of files read: 如果读取的文件数太多，而 size of files read/number of output rows 太小，可以认为小文件太多，需要合并小文件，或者使用 数据优化 来配置自动合并。
- scan time total (min, med, max)：分布式读取时的时间分布，如果存在部分任务读取太慢，可能是因为存储的 COS 桶限流或限频，建议查看 COS 桶监控，可以提工单给 COS 调整带宽。
- size of files read: 辅助判断读取的数据量是否合理。如果发现读取的文件太大，需要判断是否存储的文件格式不对，或有脏数据存在。
- number of output rows: 辅助判断读取的数据量是否合理。如果发现读取的数据量太大，需要结合 Scan 的 Filter 条件判断是否过滤条件有误导致扫描了全表。Spark引擎可以自动下推过滤条件到 Scan 算子，但如果某些特殊情况没有自动下推过滤条件，可以提前将 SQL 中的 where 下推。

第二步：寻找计算瓶颈

建议先从 Stages 页面查看耗时最长的 StageId，再根据 StageID 到执行计划（页面上需要勾选 Show the Stage ID and Task ID）页面分析。
在任务洞察中可以识别常见的性能瓶颈点，而在执行计划中分析，往往需要结合瓶颈算子的上下文来分析。

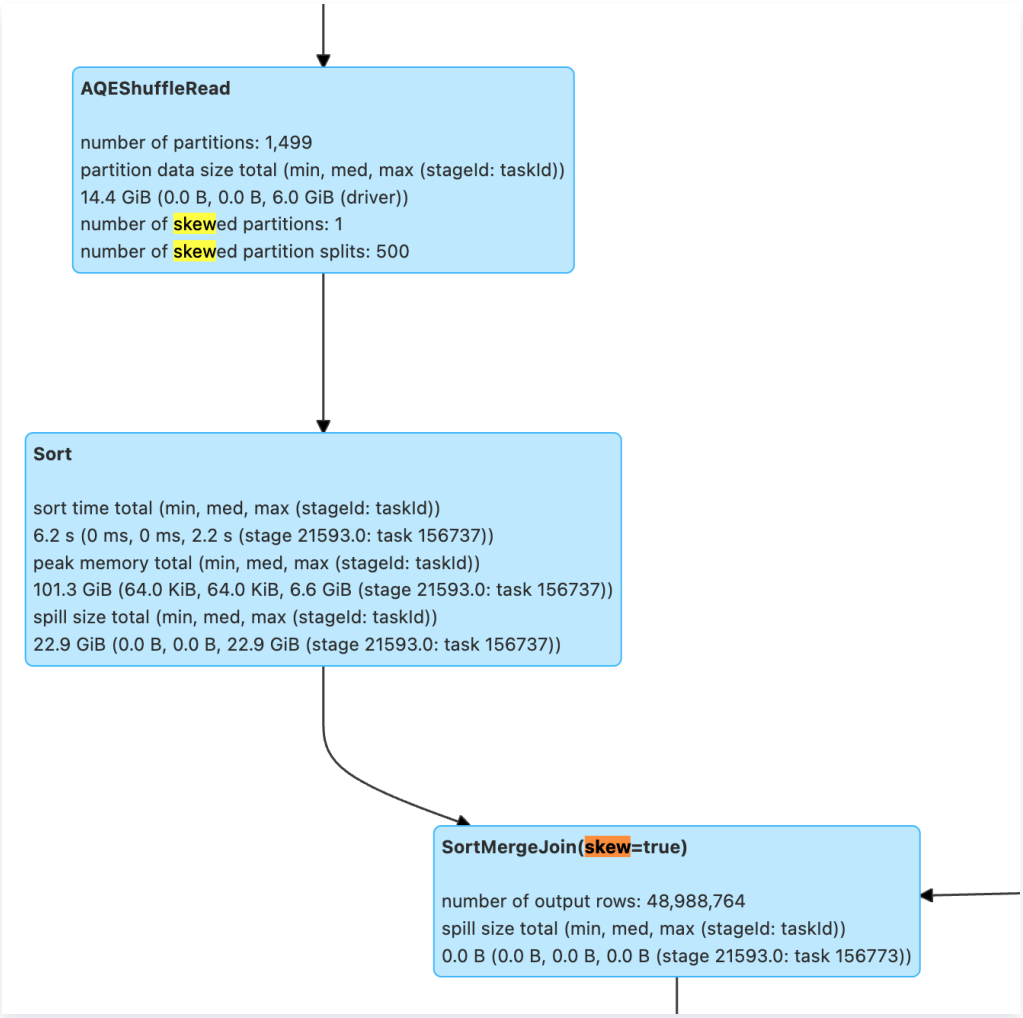
- Project 算子

  Porject 算子往往需要关注是否有大量重复的 Expression 定义，这种建议 SQL 中改写来避免大量的重复计算。
- Filter 算子

  过滤算子的过滤条件一般重点检查是否正确过滤数据，重点关注 Filter 前后数据量是否有正常的减少。
- Join 算子

  Join 是较为常见出现性能瓶颈的算子。常见的优化思路如下。

SortMergeJoin 出现 skewJoin，尽管 Spark 会默认对倾斜的 Join 做优化，但是当数据膨胀系数不大（spark.sql.adaptive.skewJoin.skewedPartitionFactor=5.0，超过平均值5倍认为是倾斜），或者当倾斜并不是特别突出（spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes=256MB，大于256MB才会认为倾斜）时一旦导致 spill 性能会急剧下降，仍然有一定的调优空间。



Skew Join示例

SortMergeJoin 数据量太大，可以结合 SQL 和执行计划分析：

1. 是否出现了笛卡尔积，join on 的条件字段存在重复，导致 left join 或 inner join 数据膨胀。
2. 是否有些过滤条件没有下推，例如 Aggregate 操作可能会隔绝过滤条件的下推，设置 spark.sql.optimizer.AggregatePushdownThroughJoins.enabled=true 可以查看是否可以下推 Aggregate 后的算子。

SortMergeJoin 出现大表 Join 小表，是否可以考虑替换为 BroadcastHashJoin。通过 Spark hints 或调大 spark.sql.autoBroadcastJoinThreshold=10MB 来让小的表可以广播。

- Aggregate 算子

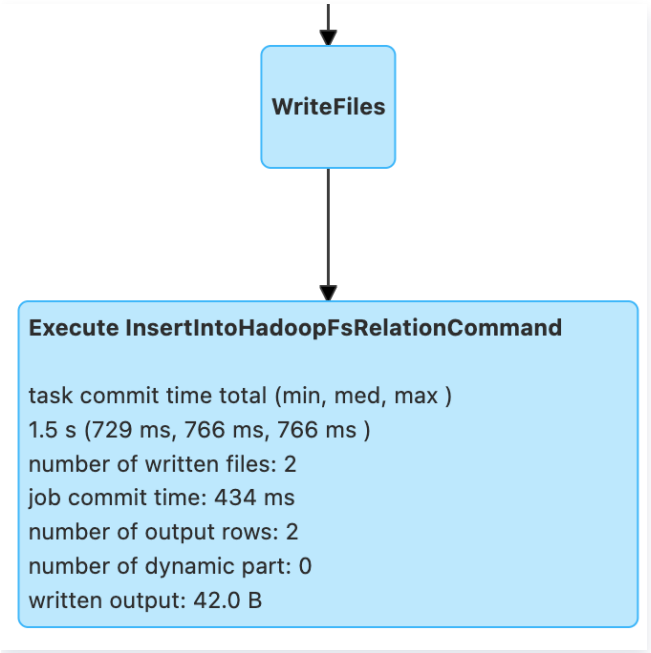
  Aggregate 比较常见的是倾斜的问题，当 aggregate key 明显过多时，会导致慢 task 的产生。如果调整分区数无法解决倾斜，一般只能通过重新调整 key 来打散过多的 Key，来重新做业务 SQL 设计。此外，如果 Aggregate 的函数存在重复计算，也可以考虑从 SQL 层面做精简。

- Exchange + AQEShuffleRead 算子

  AQE 是 Spark 提供的在 shuffle 时自动合并过小的分区、打散过大的分区的特性。上面的 Join/Aggregate 往往也会需要查看 Exchange+AQEShuffleRead 的算子。AQE 是根据 spark.sql.adaptive.advisoryPartitionSizeInBytes=64MB 来判断 ShuffleRead 时的分区大小的。所以当业务的出现数据膨胀非常多，例如小数据量 Join 产生非常大的数据量，可以考虑调小 advisory的 partition size；或者在 Shuffle 经过处理数据量缩小得非常多，可以考虑调大 advisory 的 partition size 来避免产生过多的小文件。

第三步：查看 Insert/Write 算子

Stages 页面包含 Output 的 Stage，一般可以对应到写入的算子。
一般任务结尾都为写入的算子，例如 Insertxxx/Overwritexxx/AppendData 等，常常需要关注是否有写入过多的小文件，以及 commit 的时间。



InsertIntoHadoopFsRelationCommand示例

常见的指标含义：

- number of output row：写入的数据量是否符合预期，尤其在性能变动大时需要关注。
- task/job commit time：数据写入后，需要 commit 确认整张表的数据写入正确，如果 commit 时间过长，一般是普通桶 rename 的时间过长，可以考虑换元数据加速桶，可以工单咨询 COS 桶。
- number of dynmaic part：默认动态分区写入，如果只写入了一个分区，需要查看写入算子前是否有 Exchange 算子只对某个只有一个值的字段做了分区，这会导致分布式写边成单线程写。如果是 Hive 表写入可以通过 Spark hints 调整写入时的分区策略；如果是 Iceberg 表则需要调整 `spark.sql.iceberg.distribution-mode`=none。

Presto

优化 SQL 语句

场景：SQL 语句本身不合理，导致执行效率不高。

优化 JOIN 语句

当查询涉及 JOIN 多个表时， Presto 引擎会优先完成查询右侧的表的 JOIN 操作，通常来说，先完成小表的 JOIN，再用结果集和大表进行 JOIN，执行效率会更高，因此 JOIN 的顺序会直接影响查询的性能，DLC presto 会自动收集内表的统计数据， 利用 CBO 对查询中的表进行重排序。
对于外表，通常用户可以通过 analyze 语句完成统计数据的收集，或者手动指定 JOIN 的顺序。如需手动指定请按表的大小顺序，将小表放在右侧，大表放在左侧， 如表A > B > C, 例如：select * from A Join B Join C。需要注意的是，这不能保证所有场景下都能提升效率，实际上这取决于 JOIN 后的数据量大小。

优化 GROUP BY 语句

合理安排 GROUP BY 语句中字段顺序对性能有一定提升，请根据聚合字段的基数从高到低进行排序，例如：

```
//高效的写法
SELECT id,gender,COUNT(*) FROM table_name GROUP BY id, gender;
//低效的写法
SELECT id,gender,COUNT(*) FROM table_name GROUP BY gender, id;
```

另一种优化方式是，尽可能地使用数字代替具体分组字段。这些数字是 SELECT 关键字后的列名的位置，例如上面的 SQL 可以用以下方式代替：

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

使用近似聚合函数

对于允许有少量误差的查询场景，使用这一些近似聚合函数对查询性能有大幅提升。
例如，Presto 可以使用 APPROX_DISTINCT()函数代替 COUNT(distinct x)，Spark 中对应函数为 APPROX_COUNT_DISTINCT 。该方案缺点是近似聚合函数有大概2.3%的误差。

使用 REGEXP_LIKE 代替多个 LIKE

当 SQL 中有多个 LIKE 语句时，通常可以使用正则表达式来代替多个 LIKE，这样可以大幅提升执行效率。例如：

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

可以优化成：

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

数据治理

数据治理适用场景

场景：实时写入。Flink CDC 实时写入通常采用 upsert 的方式写入，该流程在写入过程中会产生大量的小文件，当小文件堆积到一定程度后会导致数据查询变慢，甚至超时无法查询。
可以通过以下方式查看表文件数量和快照信息。

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

例如：

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

表文件、快照数量过多时，可以参考文档 开启数据治理 启用数据治理功能。

数据治理效果

开启数据治理后，查询效率得到显著提升，例如下表对比了合并文件前后的查询耗时，该实验采用16CU presto，数据量为14M，文件数量2921，平均每个文件0.6KB。

执行语句	是否合并文件	文件数量	记录条数	查询耗时	效果
SELECT count(*) FROM tb	否	2921个	7895条	32s	速度快93%
SELECT count(*) FROM tb	是	1个	7895条	2s	

分区

分区能够根据时间、地域等具有不同特征的列值将相关数据分类存储，这有助于大幅减少扫描量，提升查询效率。关于 DLC 外表分区更多详情信息，请参考一分钟入门分区表 。下表展示了在数据量为66.6GB，数据记录为14亿条，数据格式为 orc 的单表中，分区和不分区时查询耗时和扫描量的效果对比。其中`dt`是含有1837个分区的分区字段。

查询语句	未分区		分区		耗时对比	扫描量对比
	耗时	扫描量	耗时	扫描量		

SELECT count(*) FROM tb WHERE dt='2001-01-08'	2.6 s	235.9 MB	480 ms	16.5 KB	快 81%	少99.9%
SELECT count(*) FROM tb WHERE dt<'2022-01-08' AND dt>'2001-07-08'	3.8 s	401.6 MB	2.2 s	2.8 MB	快 42%	少99.3%

从上表中可以看出，分区可以有效地降低查询延时和扫描量，但过度分区可能适得其反。如下表所示。

查询语句	未分区		分区		耗时对比	扫描量对比
	耗时	扫描量	耗时	扫描量		
SELECT count(*) FROM tb	4s	24MB	15s	34.5MB	慢73%	多30%

建议您在 SQL 语句中通过 WHERE 关键字来过滤分区。

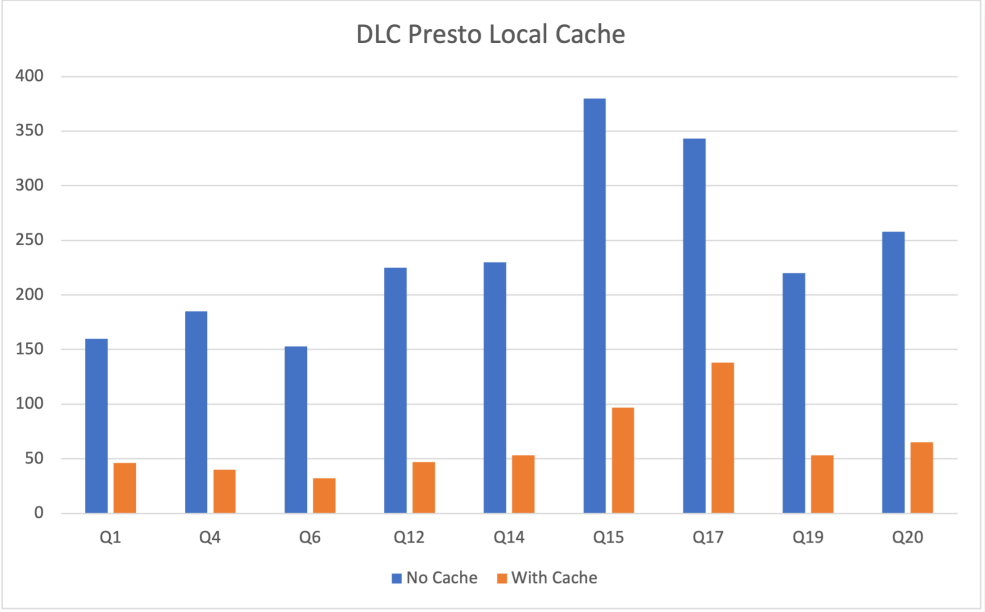
缓存

在如今分布式计算和存算分离的趋势下，通过网络访问元数据以及海量数据将会受到网络 IO 的限制。DLC 默认开启以下缓存技术大幅降低响应延时，无需您介入管理。

- Alluxio：是一种数据编排技术 。它提供缓存，将数据从存储层移动到距离数据驱动型应用更近的位置从而能够更容易被访问。Alluxio 内存至上的层次化架构使得数据的访问速度能比现有方案快几个数量级。
- RaptorX：是Presto的一个连接器。它像 Presto 一样运行在存储之上，提供亚秒级延迟。 目标是为 OLAP 和交互式用例提供统一、廉价、快速且可扩展的解决方案。
- 结果缓存：Result Cache，对于重复的同一查询进行缓存，极大提高速度和效率。

DLC Presto 引擎默认支持 RaptorX 和 Alluxio 分级缓存，在短时间内相同任务场景中可以有效地降低延时。Spark、Presto引擎均支持结果缓存。

下表是在总数据量 为1TB的 Parquet 文件中的 TPCH 测试数据，本次测试选用16CU Presto。因为测试的是缓存功能，所以主要从 TPCH 中选择 IO 占用比较大的 SQL ，涉及的表主要有 lineitem、orders、customer 等表，涉及的 SQL 为 Q1、Q4、Q6、Q12、Q14、Q15、Q17、Q19 以及 Q20。其中横坐标表示SQL语句，纵坐标表示运行时间(单位秒)。



需要注意的是，DLC Presto 引擎会根据数据访问频率动态加载缓存，所以引擎启动后首次执行任务无法命中缓存，这导致首次执行仍受网络 IO 限制，但随着**执行次数增加**，该限制明显得到缓解。如下表展示了 presto 16cu 集群三次查询的性能比较。

查询语句	查询	耗时	数据扫描量
SELECT * FROM table_namewhere udid='xxx';	第一次查询	3.2s	40.66MB
	第二次查询	2.5s	40.66MB
	第三次查询	1.6s	40.66MB

您可以在DLC控制台 数据探索 功能中查看执行的SQL任务的缓存命中情况。

索引

内表+索引的建表方式相对于外表，在时间和扫描量上均会大幅减小，关于创建表的更多详细信息，请参考 数据表管理 。

创建表后根据业务使用频率在 insert 前建立索引，WRITE ORDERED BY 后的索引字段。

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

下表展示了 presto 16cu 集群在外表和内表（加索引）上查询性能比较

表类型	查询	耗时	数据扫描量
外表	第一次查询	16.5s	2.42GB
	第二次查询	15.3s	2.42GB
	第三次查询	14.3s	2.42GB
内表（索引）	第一次查询	3.2s	40.66MB
	第二次查询	2.5s	40.66MB
	第三次查询	1.6s	40.66MB

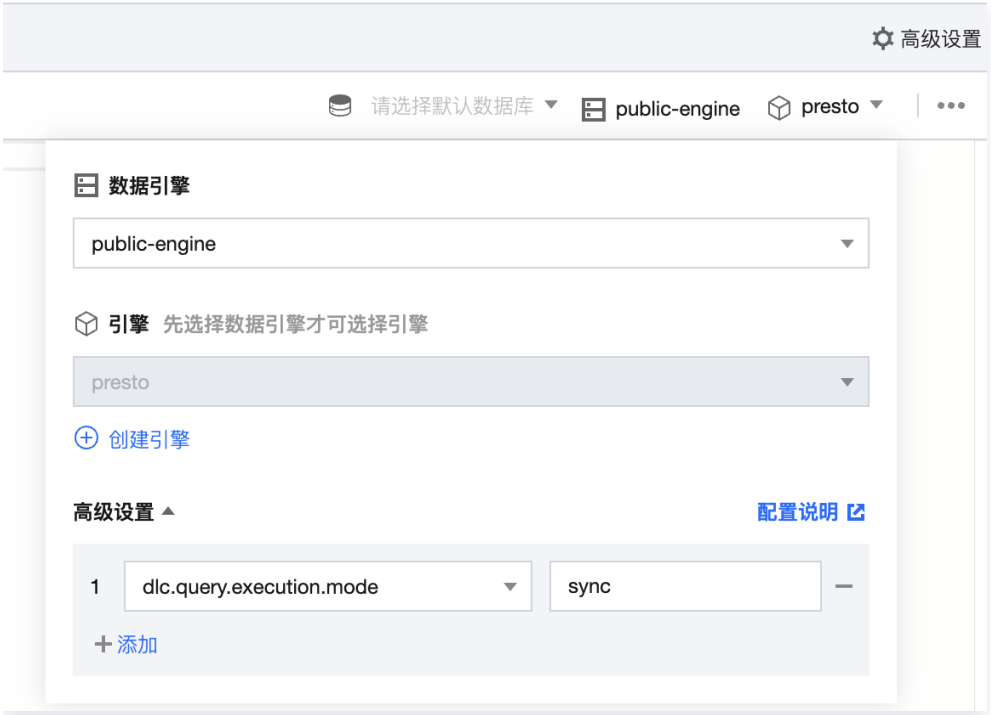
从表中可以看出，内表+索引的建表方式相对于外表，在时间和扫描量上均会大幅减小，并且由于缓存加速，**执行时间也会随着执行次数的增加而减少**。

同步查询和异步查询

DLC 针对于 BI 场景进行了特别的优化，可以通过配置引擎参数dlc.query.execution.mode来开启同步模式或者异步模式（只支持 presto 引擎）。取值介绍如下。

- async（默认）：该模式任务会完成全量查询计算，并将结果保存到 COS，再返回给用户，允许用户在查询完成后下载查询结果。
- sync：该模式下，查询不一定会执行全量计算，部分结果可用后，会直接由引擎返回给用户，不再保存到 COS。因此用户可获得更低查询延迟和耗时， 但结果只在系统中保存30s。推荐不需要从 COS 下载完整查询结果，但期望更低查询延迟和耗时时使用该模式，例如查询探索阶段、BI 结果展示。

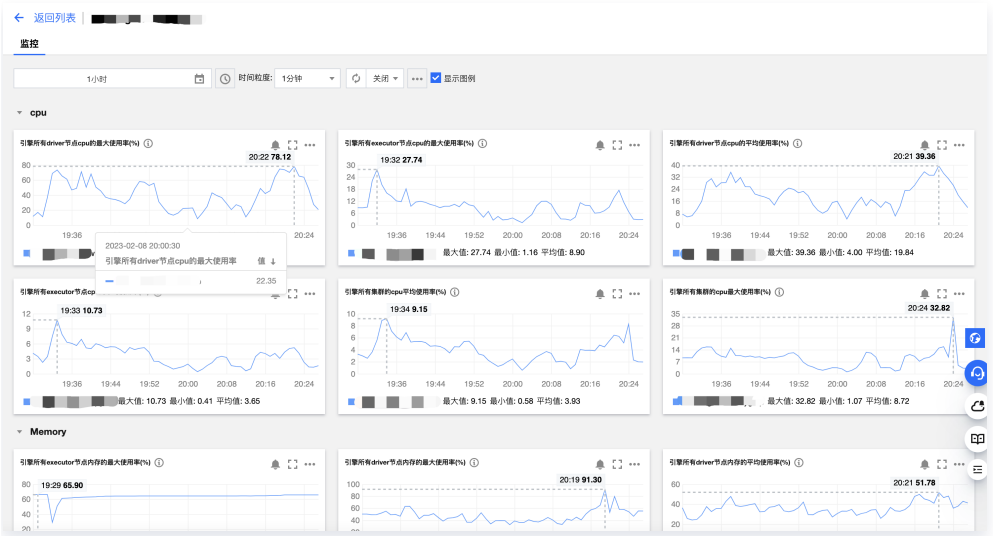
配置方式：选择数据引擎后，支持对数据引擎进行参数配置，选择数据引擎后，在高级设置单击添加即可进行配置。



资源瓶颈

评估资源是否达到瓶颈，DLC 提供引擎的 CPU、内存、云盘、网络等资源监控。您可以根据业务规模调整资源规格，变配请参考 调整配置费用说明 。查看引擎资源使用情况步骤如下：

1. 打开左侧数据引擎标签页。
2. 单击相应引擎的右侧**监控**按钮。
3. 跳转到腾讯云可观测平台，可以查看到所有监控指标，如下图所示。详细操作以及监控指标请参考数据引擎监控。同时您也可以针对每个指标进行告警配置，详细介绍请参考 监控告警配置 。



其他因素

自适应 shuffle

DLC 默认关闭自适应 shuffle。当您任务出现 No space left 等磁盘空间不足的问题时，可以开启自适应 shuffle，这是一套即能支持有限本地磁盘的常规 shuffle，又能保证在大 shuffle 和数据倾斜等场景下的稳定性。

自适应 shuffle 带来的优势：

1. 降低存储成本：集群节点的磁盘挂载量进一步降低，一般规模集群每节点只要50G、大规模集群也不超200G。
2. 稳定性：对于 shuffle 数据量剧增或数据倾斜场景任务执行的稳定性不会再因本地磁盘限制而失败。

尽管自适应 shuffle 带来存储成本的降低和稳定性提升，但在某些场景下，如资源不足时，会带来约15%的延时。

【开启方式】

集群配置中新增以下配置开启：

spark.shuffle.manager=org.apache.spark.shuffle.enhance.EnhancedShuffleManager

可选配置：

如果您希望提前写入数据到远端，确保磁盘有足够空间，避免影响其他任务，可以考虑以下配置：

spark.shuffle.enhanced.usage.waterLevel1=磁盘比例，shuffle 结果写到远端时磁盘已使用的比例，默认0.7。

spark.shuffle.enhanced.usage.waterLevel2=磁盘比例，shuffle spill 数据写到远端时磁盘已使用的比例，默认0.9。

集群冷启动

DLC 支持自动或者手动挂起集群，挂起后不再产生费用，所以在集群启动后，首次执行任务可能存在“正在排队”的提示，这是因为集群冷启动中正在拉起资源。如果您频繁提交任务，建议购买包年包月集群，该类型集群不存在冷启动，能在任何时间快速执行任务。

UDF 函数开发指南

最近更新时间：2025-04-29 15:18:12

UDF 说明

用户可通过编写 UDF 函数，打包为 JAR 文件后，在数据湖计算定义为函数在查询分析中使用。目前数据湖计算 DLC 的 UDF 为 HIVE 格式，继承 org.apache.hadoop.hive.ql.exec.UDF，实现 evaluate 方法。
示例：简单数组 UDF 函数。

```
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;
    }
}
```

pom 文件参考：

```
<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>
```

创建函数

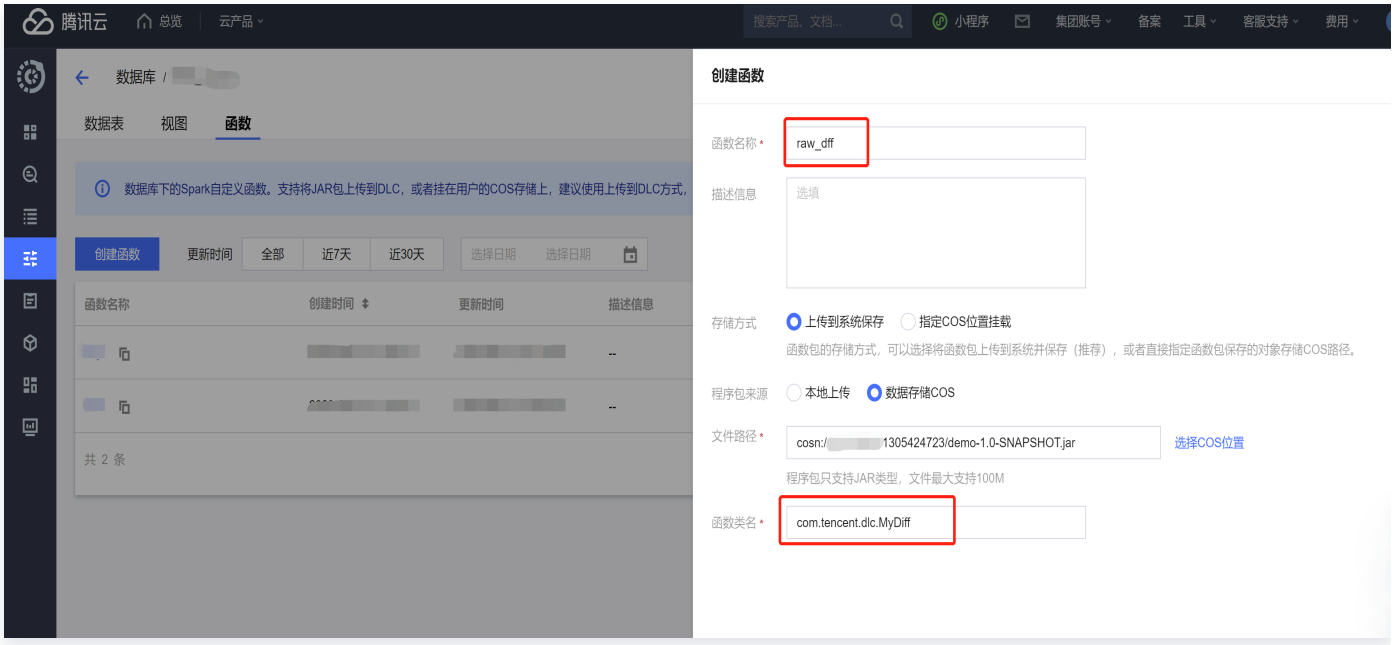
> ⚠ 注意：

> 如您创建的是 udaf/udtf 函数，需要在函数名相应加上 _udaf/_udtf 后缀。

若您了解 SQL 语法，可通过**数据探索**执行 CREATE FUNCTION 语法完成函数创建，或通过可视化界面创建，流程如下：

1. 登录 数据湖计算控制台 ，选择服务地域。
2. 通过左侧导航菜单进入**数据管理**，选择需要创建的函数的数据库，如果需要创建新的数据库，可参见 数据库管理 。

3. 单击**函数**进入函数管理页面。
4. 单击**创建函数**进行创建。



UDF 的程序包支持本地上传或选择 COS 路径（需具备 COS 相关权限），示例为选择 COS 路径创建。
函数类名包含“包信息”及“函数的执行类名”。

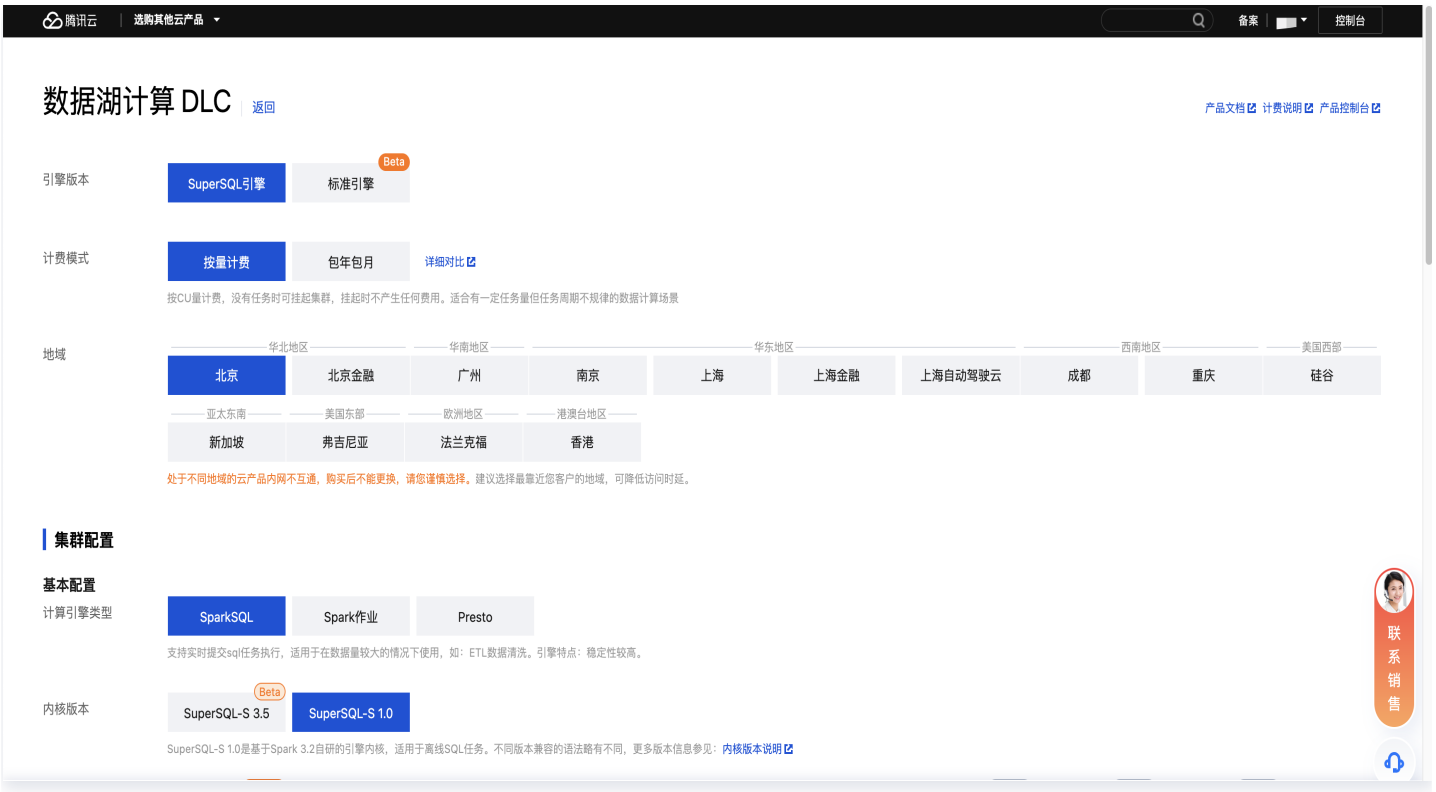
函数使用

1. 登录 数据湖计算控制台，选择服务地域。
2. 通过左侧导航菜单进入数据探索，选择计算引擎后即可使用 SQL 调用函数（注意：请选择独享引擎，共享引擎暂不支持自定义 udf 函数使用）。

Python UDF 开发

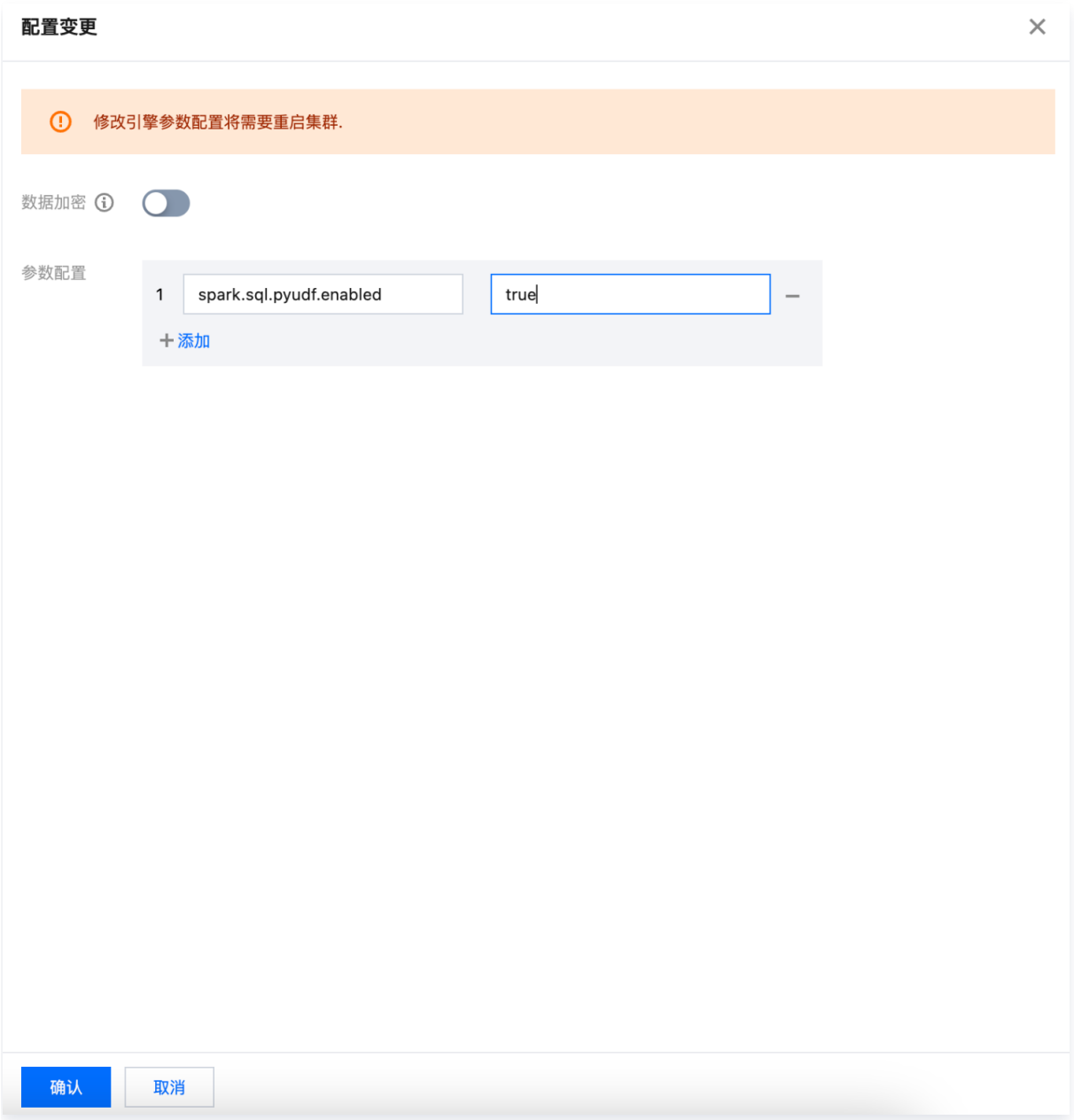
如需使用 Python UDF 功能，请将集群升级到最新版本，您可以 提交工单 咨询我们当前集群是否支持。

1. 购买集群，请选择 SuperSQL-S 1.0 版本（暂支持该版本）集群。



2. 开启集群 Python UDF 功能

- 默认情况下，Python UDF 功能未开，请在 SuperSQL 引擎页面，选择需要开启的引擎，单击**参数配置**，加入配置。输入 `spark.sql.pyudf.enabled = true` 后，单击**确认**，进行保存。
- SparkSQL 集群待集群重启后生效。
- SparkBatch 集群下一个任务生效。



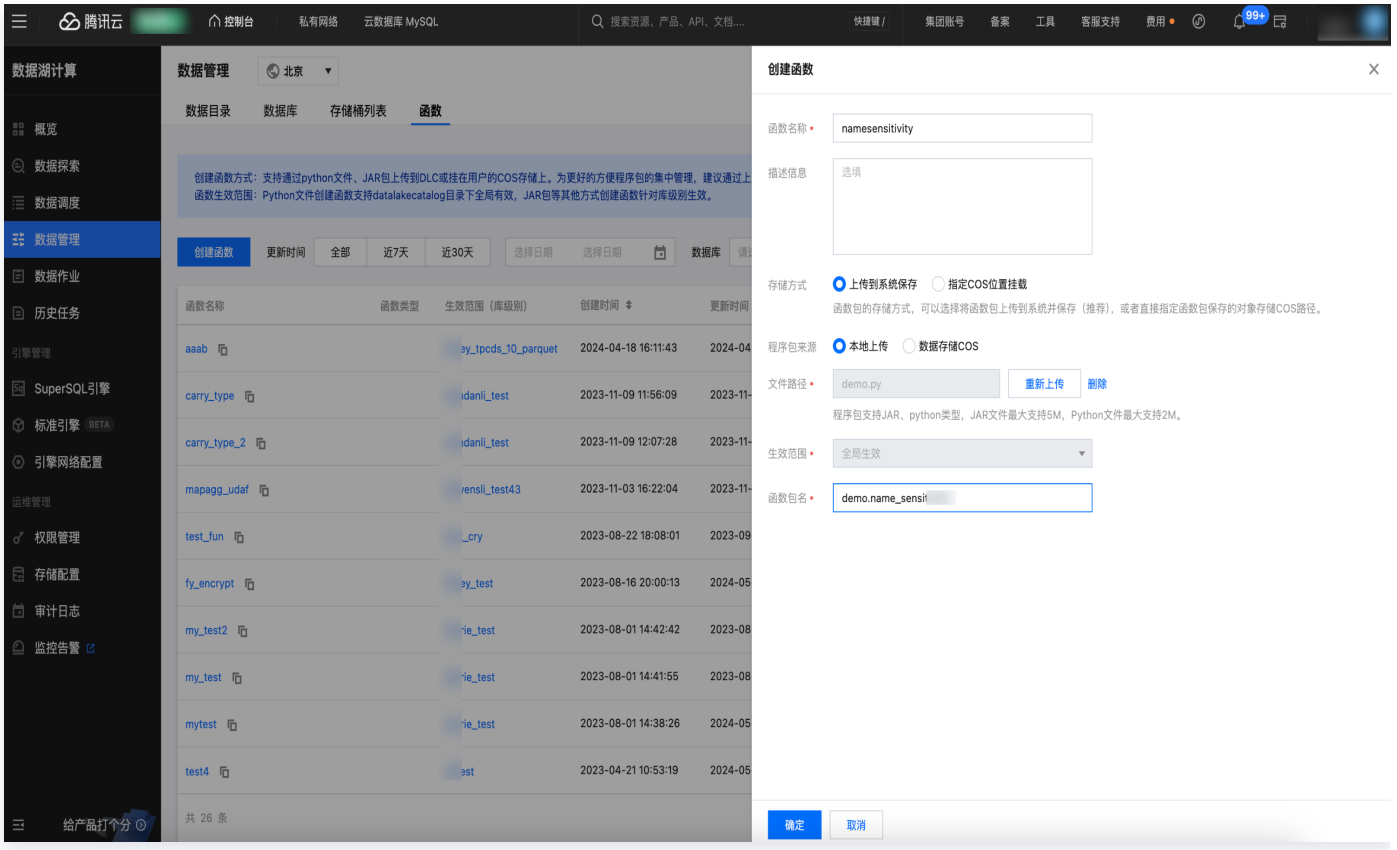
3. 注册 Python UDF 函数

- DLC 支持完全社区兼容的 UDF 函数。

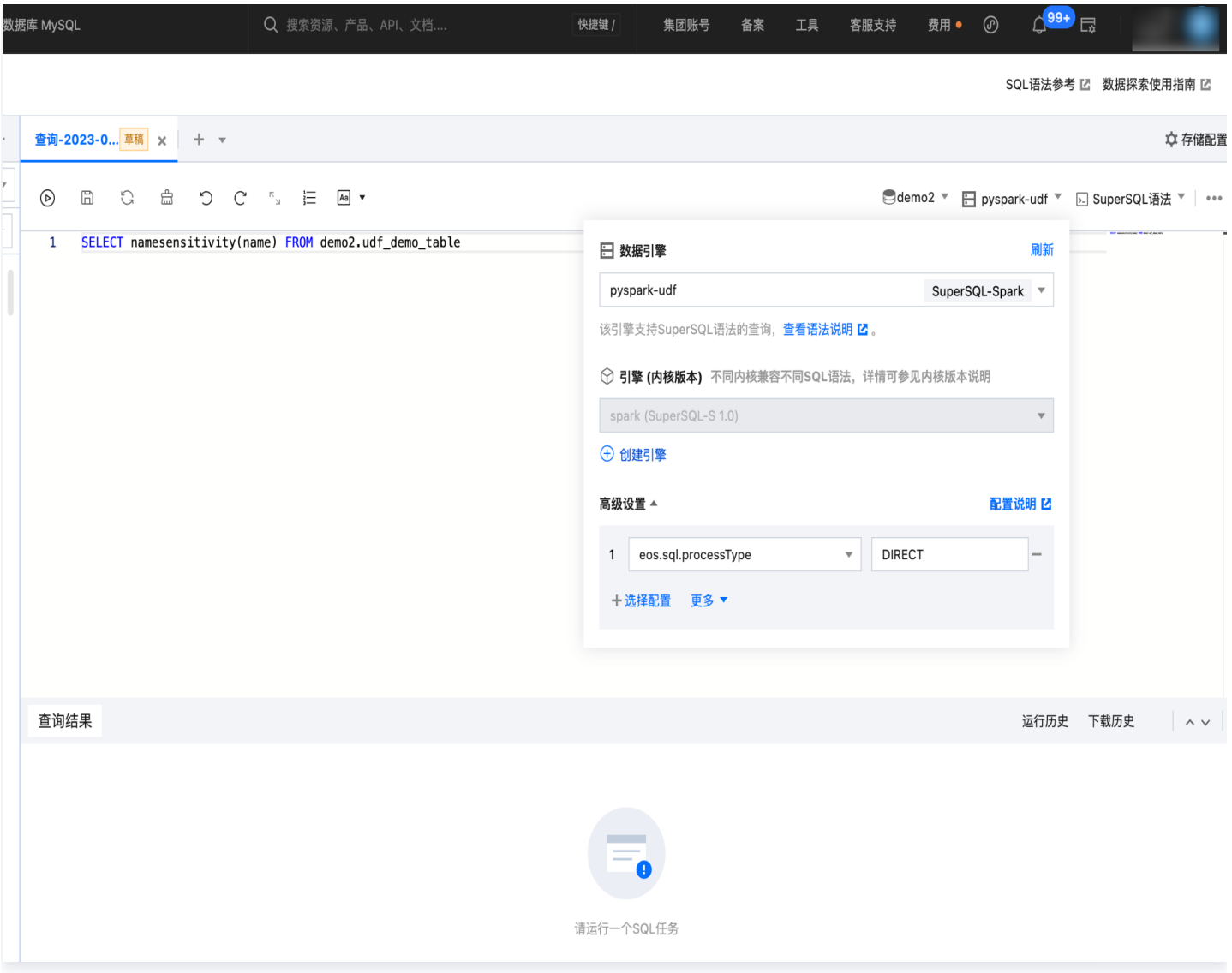
  示例，请编辑如下文件 `demo.py` ：

```
def name_sensitivity(name):
    if len(name) > 2:
        return name[0] + '*' * (len(name) - 2) + name[-1]
    elif len(name) == 2:
        return '*' + name[-1]
    else:
        return name
```

- 目前支持单 Python 文件。
- 只支持 import 系统自带模块。
- 请注意函数输入、输出类型。
- 请注意处理函数异常。



4. 在数据探索界面，使用开启了支持 Python UDF 的集群，并设置 Session 参数 `eos.sql.processType=DIRECT` 如果您不能编辑该参数，请 提交工单 联系我们。



- 如果您需要编辑变更 Python UDF 函数，SparkSQL 集群加载机制约有30s延迟，SparkBatch 集群下一个任务立即生效。
- 如果您需要删除当前 Python UDF 函数，SparkSQL 集群需要重启集群，SparkBatch 集群下一个任务立即生效。

Python 函数编辑权限配置

> **说明：**
> 在 数据湖 DLC 控制台 > 元数据管理 > 函数，点击**函数名称**进入函数详情页，仅支持查看，不支持修改。

当通过 Python 创建函数时，允许创建者及管理员对函数进行编辑权限配置。流程如下：

入口一：创建函数时进行编辑权限配置

1. 登录数据湖 DLC 控制台，选择服务地域。
2. 左侧菜单栏进入元数据管理，选择**函数**页面，点击函数栏左上方蓝色按钮“**创建函数**”，进入页面。
3. 点击“**函数编辑权限**”右侧“**展开**”，创建者可选择用户或工作组（可组合选择）进行对函数“**编辑**”和“**删除**”的授权，创建者和管理员默认拥有全部权限。
4. 设置完成后，点击“**确定**”，即可完成函数编辑权限配置。

入口二：对已有函数进行编辑权限配置

1. 登录数据湖 DLC 控制台，选择服务地域。
2. 左侧菜单栏进入元数据管理，选择**函数**页面，选择需要编辑的函数右侧**操作栏**的**编辑**按钮，即可进入编辑页面。
3. 进入编辑函数页面后，点击“**函数编辑权限**”右侧“**展开**”，可以进行权限的修改与删除。

物化视图

最近更新时间：2023-06-20 14:58:41

> **注意：**
> 目前数据湖计算 DLC 物化视图只支持 SparkSQL 引擎和 Presto 引擎。

物化视图（Materialized View）是数据库中的一种特殊对象，它是一个预先计算和存储的查询结果集。物化视图在处理大量数据和复杂查询时可以提供快速的查询性能。
物化视图提高查询性能的同时也引入了存储成本和计算成本。我们建议您在以下场景使用物化视图：

- 源表变更不频繁
- 相比于源表，物化视图表的字段和结果数量有明显的减少

DLC 支持普通物化视图和映射物化视图，以下是介绍和完整的使用示例，支持的语法列表可以参考 物化视图语法 。

普通物化视图

普通物化视图的基本使用流程包括创建、刷新、使用。
以下基于 Presto 引擎操作举例完整流程。

准备数据

执行 SQ L创建库表，并插入数据。以下语句创建了一个名为 `student` 的表。

```
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;
```

创建普通物化视图

使用 `CREATE MATERIALIZED VIEW` 语句来创建物化视图。指定物化视图的名称和查询语句，可以选择性地指定查询的来源表和条件。
以下例子，使用了一个简单的 SELECT 语句从表 `student` 中选择所有分数，并对它们进行求和操作。然后将这个求和结果作为物化视图 `mv_student_sum` 的内容。

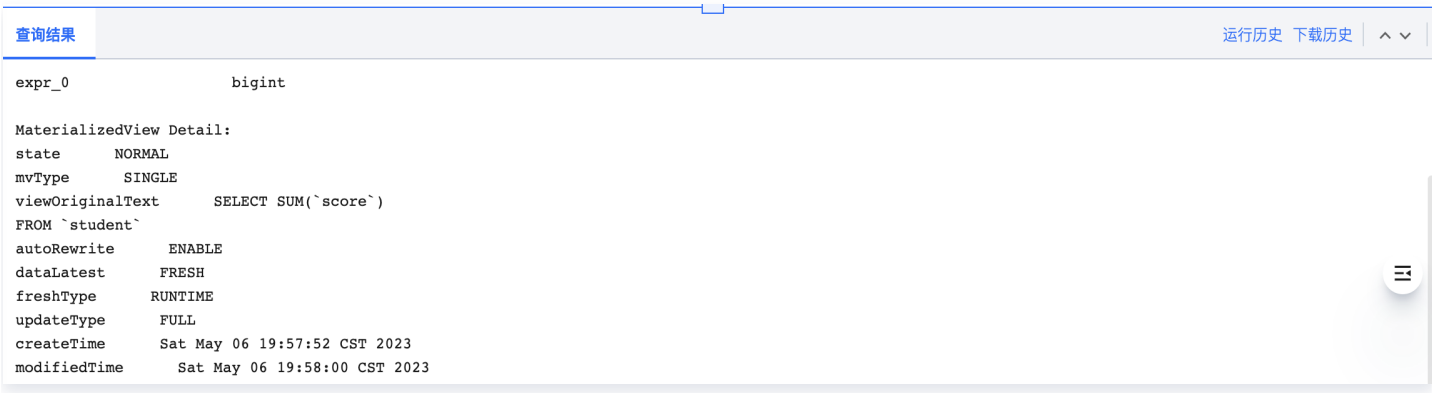
```
CREATE MATERIALIZED VIEW mv_student_sum AS (
  select sum(score) from student
```

```
);
```

查看物化视图详情

使用 `DESCRIBE MATERIALIZED VIEW` 语句来查看物化视图的详细信息，包括名称、查询语句和刷新状态等。

```
DESCRIBE MATERIALIZED VIEW mv_student_sum;
```



手动刷新物化视图

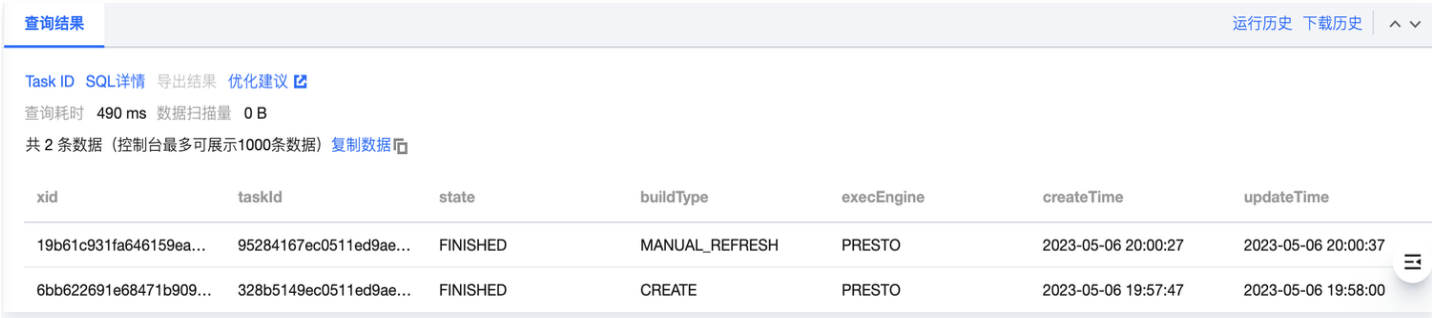
使用 `REFRESH MATERIALIZED VIEW` 语句来手动刷新物化视图的数据。
此处仅作演示，大部分情况下，您并不需要手动刷新物化视图，只要 SQL 命中了源表有变更的物化视图就会自动刷新。

```
REFRESH MATERIALIZED VIEW mv_student_sum;
```

查看物化视图的执行任务列表

使用 `SHOW MATERIALIZED VIEW JOBS` 语句来查看物化视图的执行任务列表，可以了解到物化视图的刷新历史和状态。

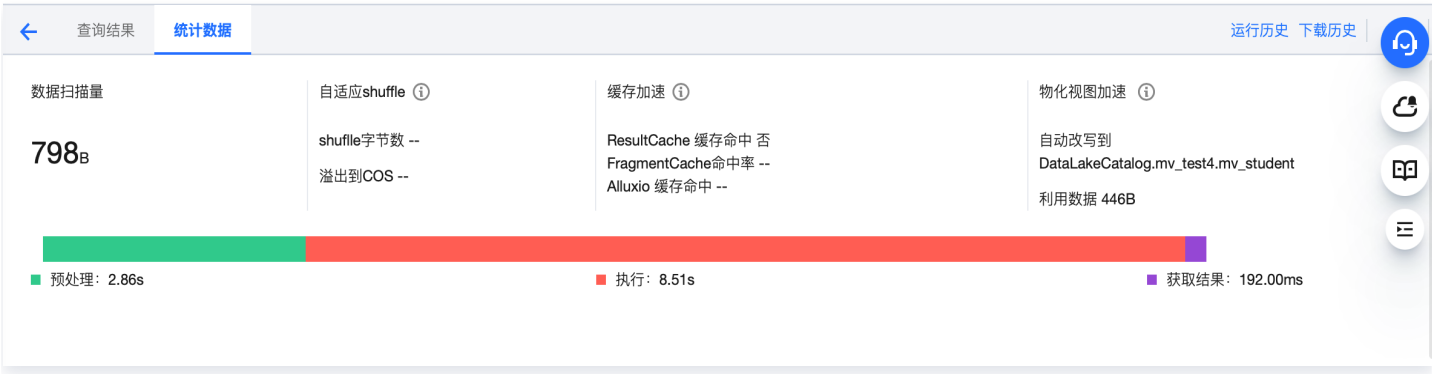
```
SHOW MATERIALIZED VIEW JOBS IN mv_student_sum;
```



SQL 改写执行

使用 SELECT 语句查询数据，期望自动改写并命中物化视图。可以通过查询结果里的统计数据，查看是否自动改写到了物化视图上。

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



删除物化视图

```
DROP MATERIALIZED VIEW mv_student_sum;
```

映射物化视图

映射物化视图是一种特殊类型的物化视图，它与现有的表进行映射关联。通过映射物化视图，可以将物化视图的查询结果与现有表的数据进行关联，从而实现对现有表的查询性能优化。

限制

物化视图相对于普通物化视图有以下限制：

- 映射物化视图不支持刷新操作，即无法通过REFRESH MATERIALIZED VIEW语句来刷新物化视图的数据。因此，物化视图的数据只能与映射表的数据保持一致，无法自动更新。
- 映射物化视图不进行自动SQL改写，即查询语句不会自动转换为使用物化视图。需要手动指定使用物化视图的查询语句。
- 删除映射物化视图时，只会删除与映射表的关联关系，而不会删除映射表本身。映射表仍然存在，可以继续使用。

推荐场景

推荐您在以下场景使用映射物化视图：

- 当已经存在一个数据量较大的表，并且该表的查询性能较低时，可以通过映射物化视图来优化查询性能。
- 当需要保持物化视图的数据与现有表的数据保持一致，并且不需要自动刷新物化视图时，可以使用映射物化视图。

Iceberg 类型的源表

Iceberg 表为源表时，完整示例如下：

基于 CTAS 创建映射物化视图

映射物化视图需要与待映射的表保持名称一致。以下例子先基于CTAS创建表，用于映射MV的创建。数据的准备可以参考普通物化视图中完整示例中的数据准备一节。

```
CREATE TABLE link_mv_student AS (
  select sum(score) from student
);
--创建映射物化视图：使用CREATE MATERIALIZED VIEW语句创建映射物化视图。
--在创建物化视图时，使用WITH META LINK子句，并指定映射表的名称作为关联。
CREATE MATERIALIZED VIEW link_mv_student WITH META LINK AS (
  select sum(score) from student
);
```

查看映射物化视图

使用 DESCRIBE MATERIALIZED VIEW 语句可以查看映射物化视图的详细信息，包括名称、查询语句和刷新状态等。

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

映射物化视图不支持刷新操作

映射物化视图不支持 REFRESH 操作，即无法通过 REFRESH MATERIALIZED VIEW 语句来刷新物化视图的数据。因此，物化视图的数据只能与映射表的数据保持一致，无法自动更新。

SQL 改写

映射物化视图不会自动对查询语句进行 SQL 改写。
如执行 `select sum(score) from student;` 不会命中映射物化视图。
可以通过使用 Hint 或 TaskConf 参数来指定允许基于映射物化视图进行 SQL 改写。

```
--手动指定需要改写SQL
select /*+ OPTIONS('eos.sql.materializedView.enableRewrite'='true') */
sum(score) from student;
```

删除映射物化视图

使用 DROP MATERIALIZED VIEW 语句来删除映射物化视图。删除映射物化视图后，仅会删除与映射表的关联关系，映射表本身仍然存在。

```
DROP MATERIALIZED VIEW link_mv_student;
DESCRIBE link_mv_student; --可查看源表还存在
```

Hive 类型的源表

Hive 表为源表时，完整示例如下：

准备初始化数据

首先，需要准备初始化数据并创建Hive基表。使用 CREATE EXTERNAL TABLE 语句创建 Hive 基表，并通过 INSERT 语句手动插入数据。

```
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;
```

创建被映射的 Hive 外表

使用 CREATE EXTERNAL TABLE 语句创建一个被映射的 Hive 外表。

```
CREATE EXTERNAL TABLE link_mv_student_hive (
```

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

向映射表插入数据，使用 INSERT OVERWRITE 语句将查询结果插入到映射表中，确保映射表的数据与 Hive 基表的数据保持一致。

```
--向映射表插入数据
INSERT OVERWRITE link_mv_student_hive
select sum(score) from student;
```

基于 Hive 外表创建映射物化视图

使用 CREATE MATERIALIZED VIEW 语句创建映射物化视图。在创建物化视图时，使用 WITH META LINK 子句，并指定上述 Hive 外表的名称作为关联。

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

系统约束
元数据信息

最近更新时间：2023-12-27 20:28:21

数据库、数据表、属性列、分区的个数。

项目	最大数量
每个账户的数据库数量	1,000
每个账户的数据表数量	10,000
每个数据库的表数	4,096
每个数据表的列数	4,096
每个表的分区数	100,000
每个主账户的分区数	1,000,000
每张表字段数量上限	4096
每个账户的自定义函数数量	100
可创建 catalog 数量	20

数据库

- 名称：最多不超过127个字符，同一个数据链接下，不允许有相同的数据库名称。
- 描述：最多不超过2048 个字符。
- 外表的数据地址（COS 地址）：888个字符（COS 路径长度限制）。
- 参数：`Map<string:string>` 的形式，每个参数的长度限制为127个字符，总长度限制为3000个字符。

数据表/视图

- 名称：最多不超过127个字符，同一个数据库下，不允许有相同的数据表名称。
- 描述：最多不超过1000个字符。
- 外表的数据地址（COS 地址）：最多不超过888个字符（COS 路径长度限制）。
- 参数：`Map<string:string>` 的形式，每个参数的长度限制为127个字符，总长度限制为512000个字符。

属性列

- 名称：最多不超过127个字符，同一个数据表下，不允许有相同的属性列名称。

- 描述：最多不超过256个字符。
- 类型：最多不超过131072个字符。若超过将无法创建。

分区

- 分区字段名称：最多不超过127个字符。

计算任务

最近更新时间：2022-05-23 17:15:01

单条 SQL 语句大小限制为2MB。