数据湖计算

开发指南

产品文档

腾讯云

文档目录

开发指南
SparkJar 作业开发指南

最近更新时间：2024-07-31 18:02:45

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。

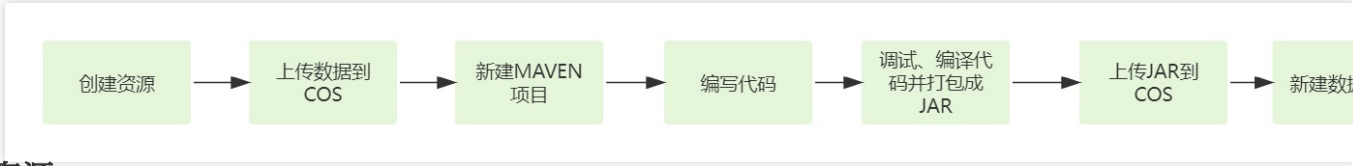
环境准备

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：

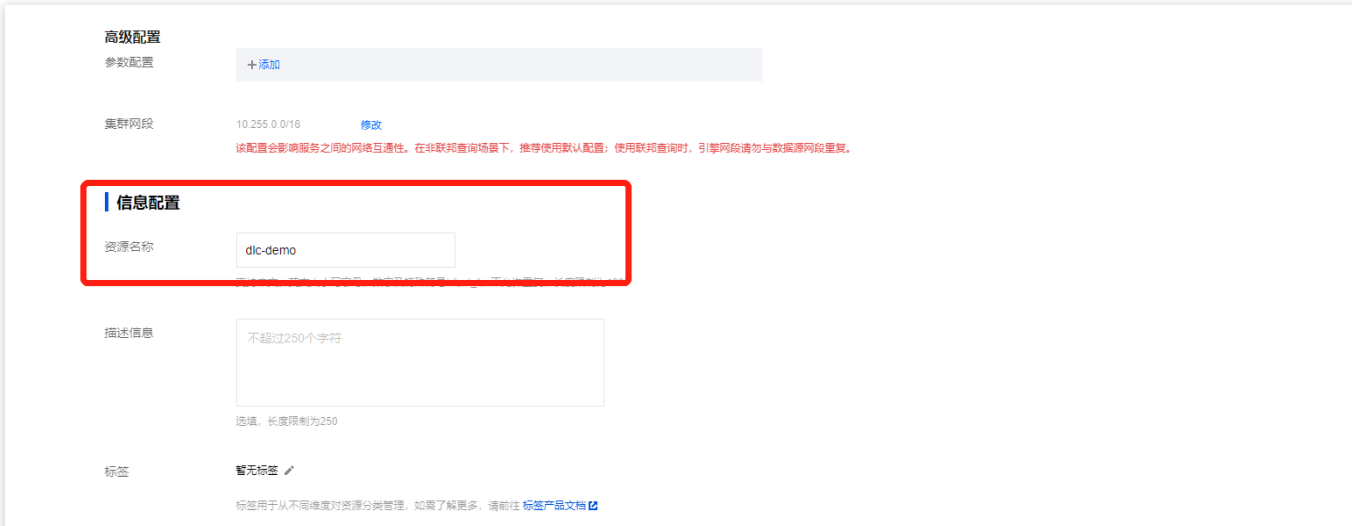


创建资源

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。

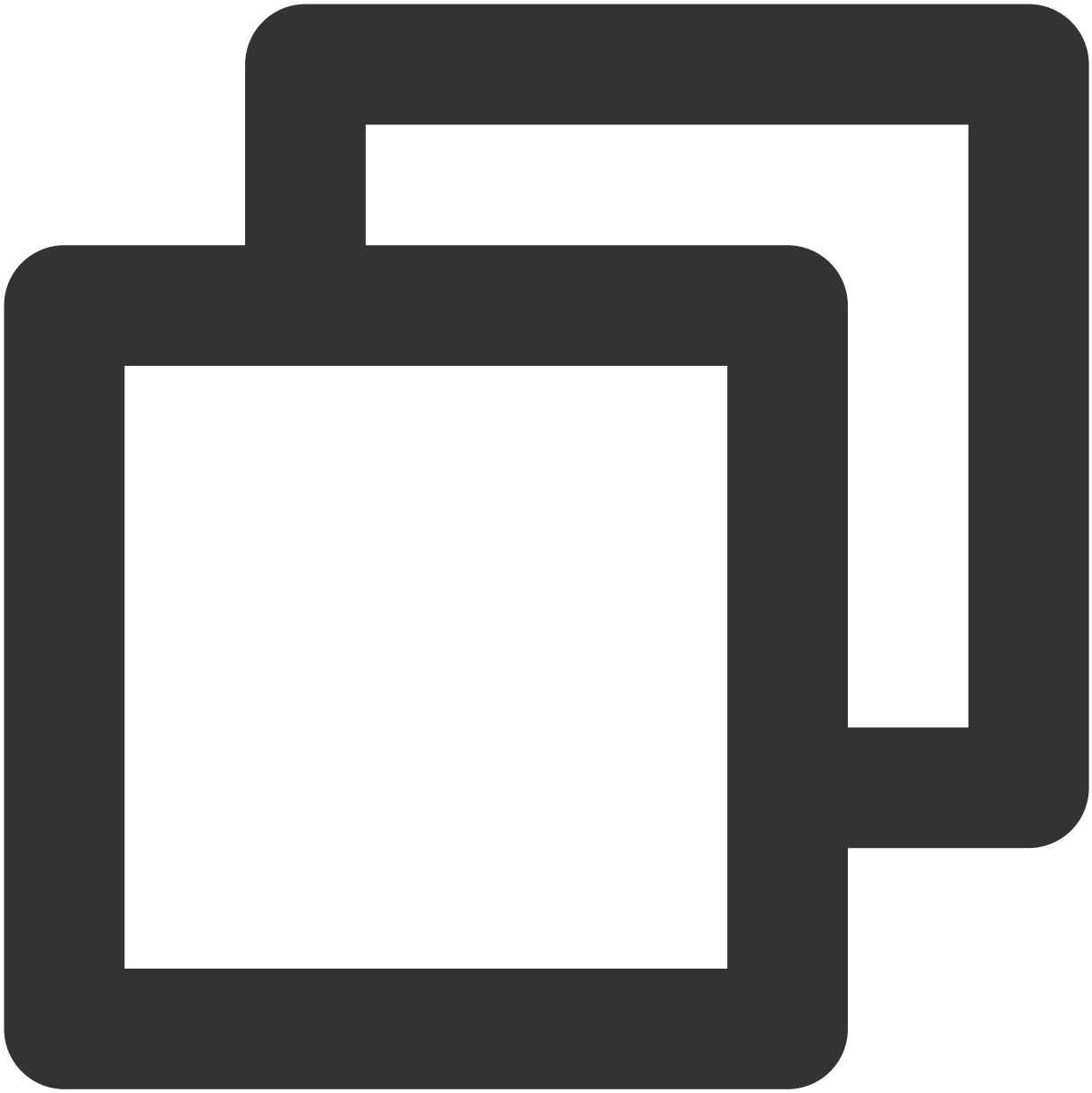
信息配置 > 资源名称 项填写 “dlc-demo”。新建资源详细介绍请参见购买独享数据引擎。



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

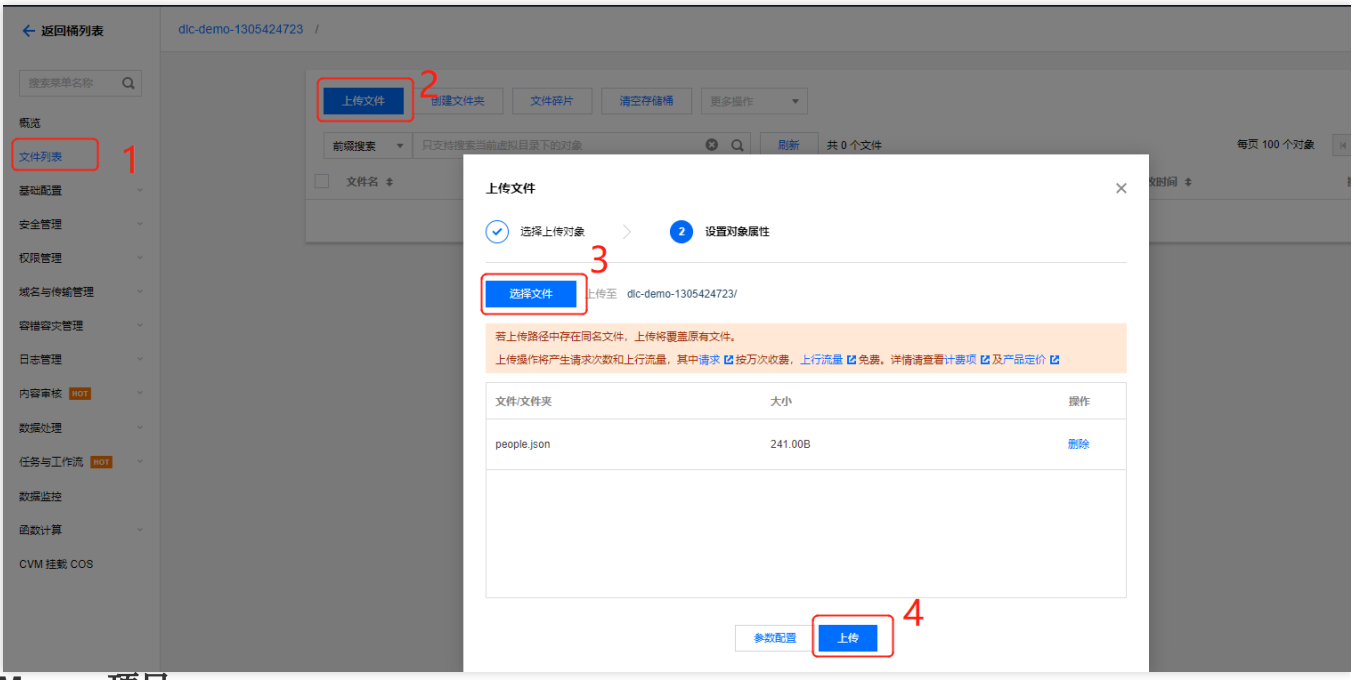
创建名称为 “dlc-demo”的存储桶，上传 people.json 文件，用作从 COS 读写数据的示例，people.json 文件的内容如下：



```
{"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}
```

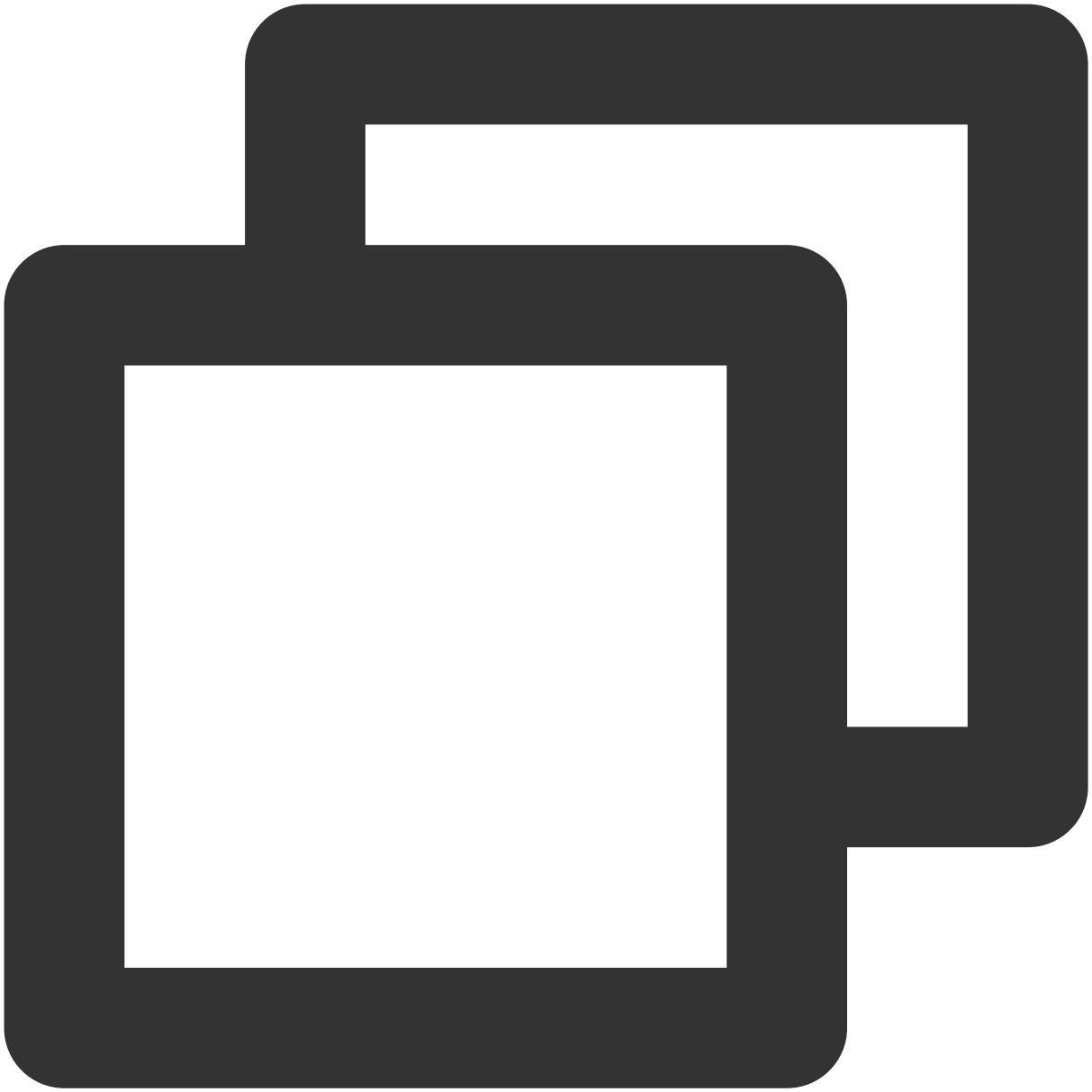
1. 登录 对象存储 COS 控制台，在左侧菜单导航中单击 **存储桶列表。**

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：



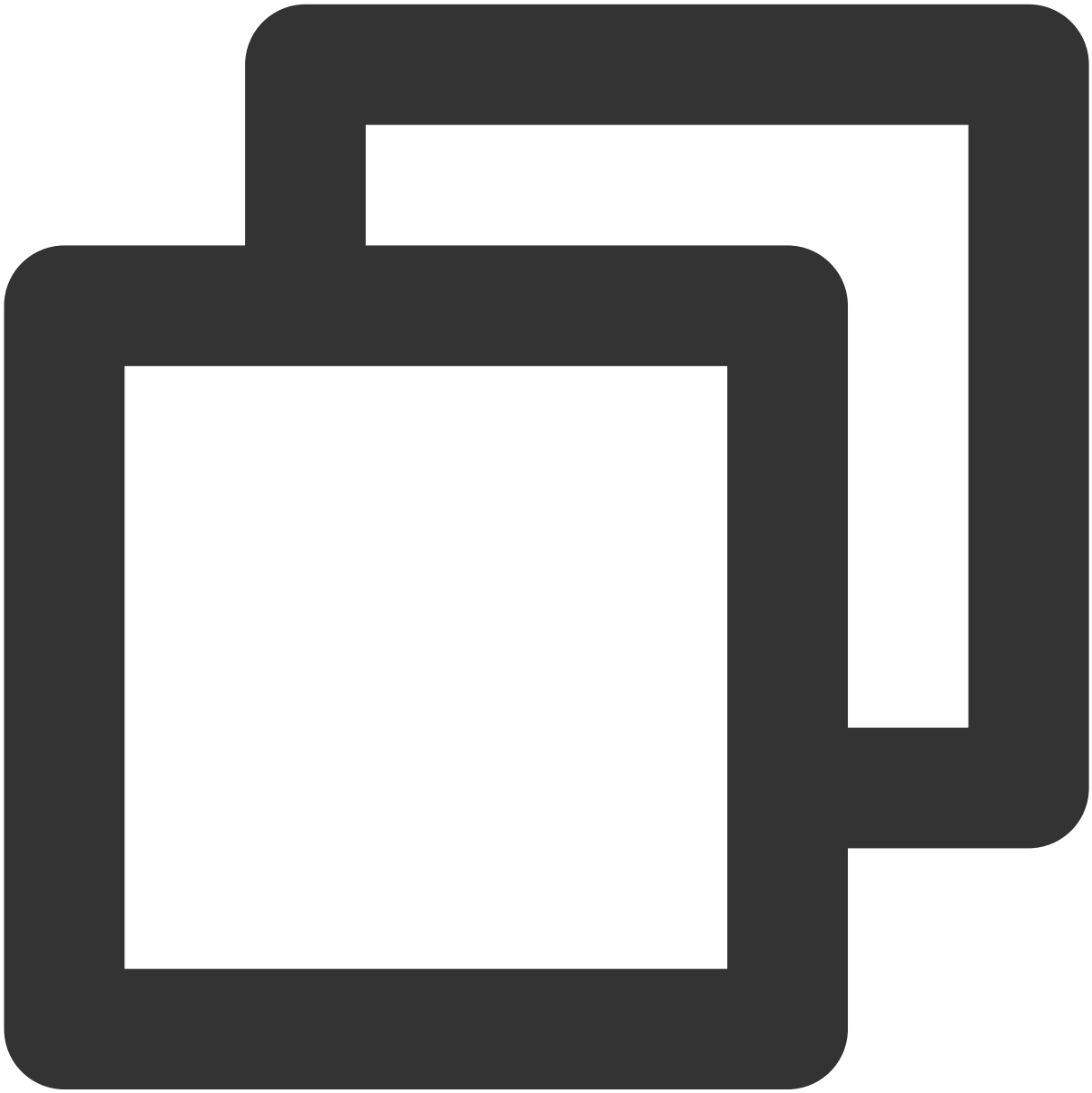
```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.12</artifactId>
  <version>3.2.1</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.12</artifactId>
  <version>3.2.1</version>
  <scope>provided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

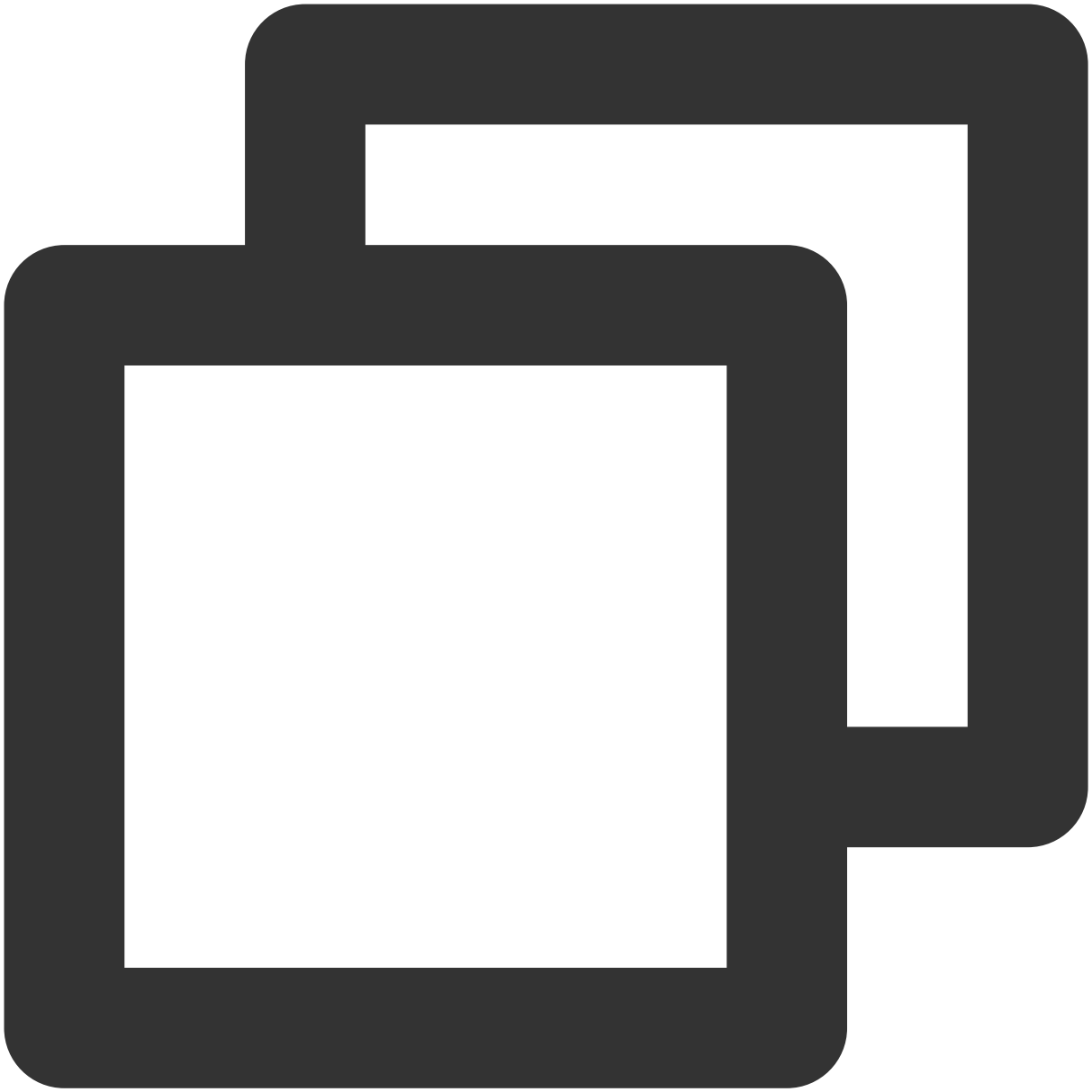
1. 从 COS 上读写数据代码示例：



```
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();
    }
}
```

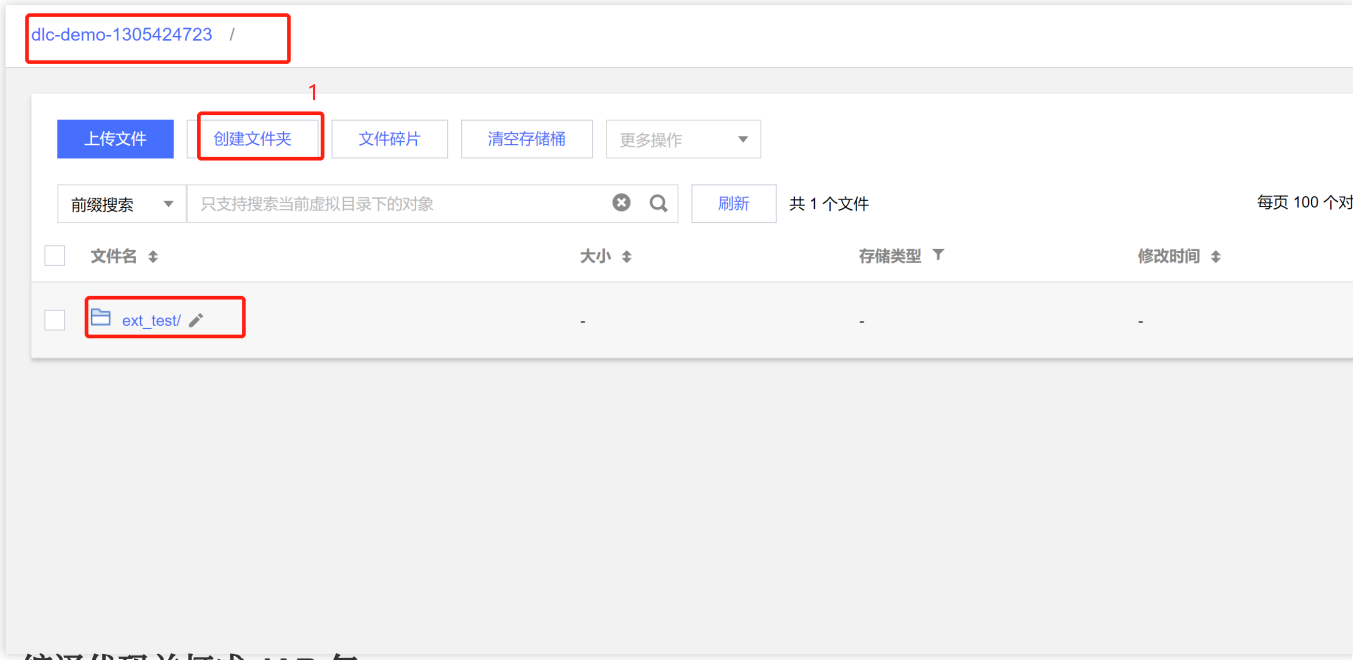
2. DLC 上建库、建表、查询数据和写入数据：



```
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 t
        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)
        //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 + "." + extTableNam
                        + "  (`id` int, `name` string, `age` int) "
                        + "ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerD
                        + "STORED AS TEXTFILE LOCATION 'cosn://dlc-demo-1305424723/
        //7.写外表数据
        spark.sql("INSERT INTO " + dbName + "." + extTableName + "VALUES (1,'LiLy',
        //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



调式、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

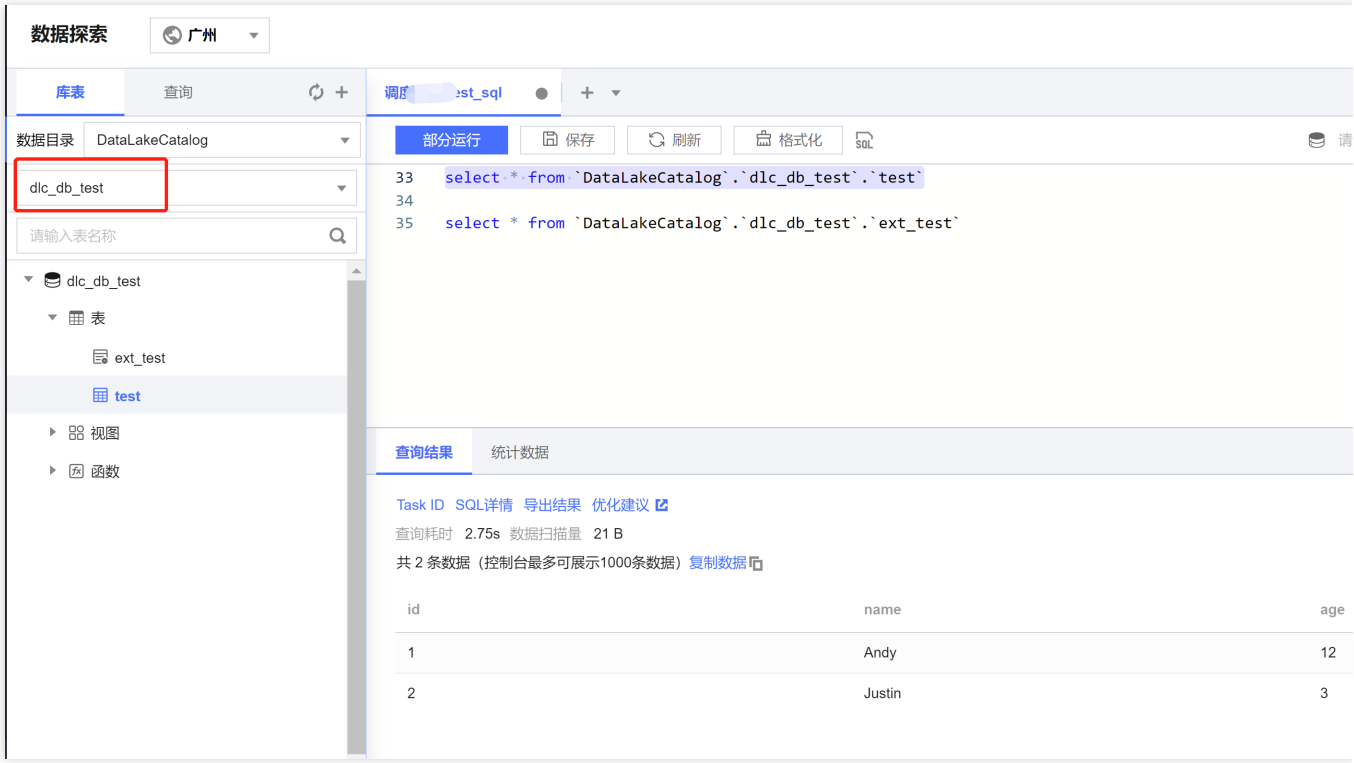
1. 单击**作业名称 > 历史任务**，查看任务运行状态。

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：2024-07-31 18:03:01

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。

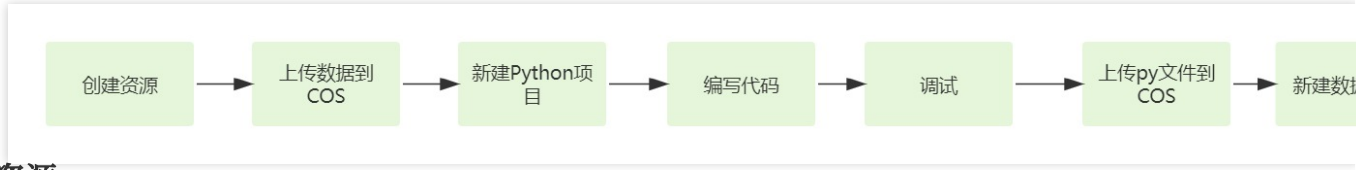
环境准备

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：

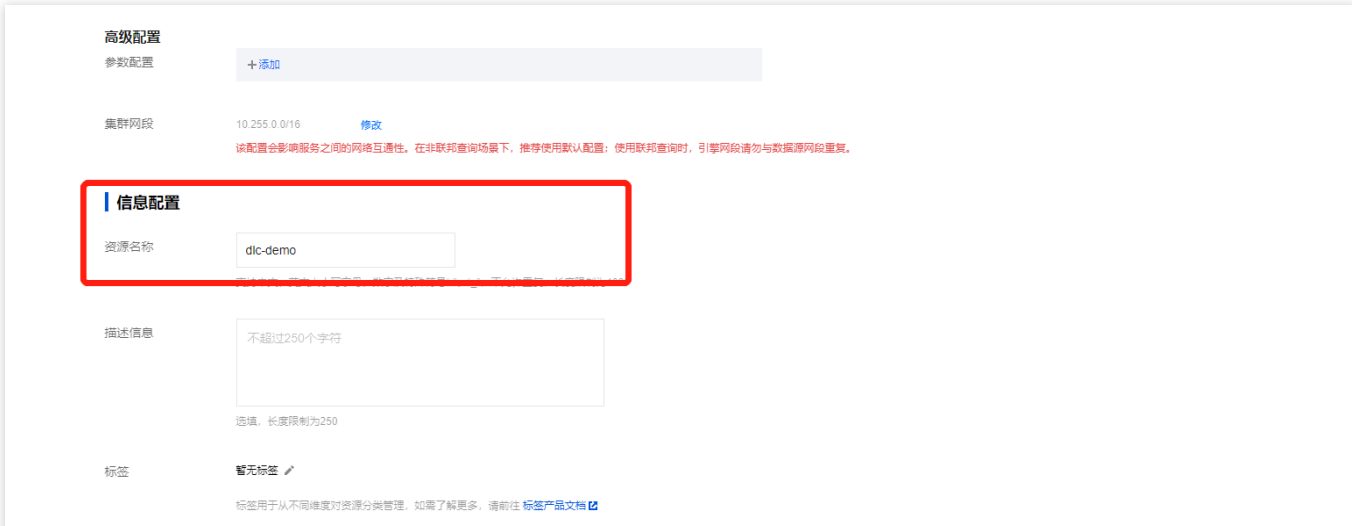


创建资源

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。

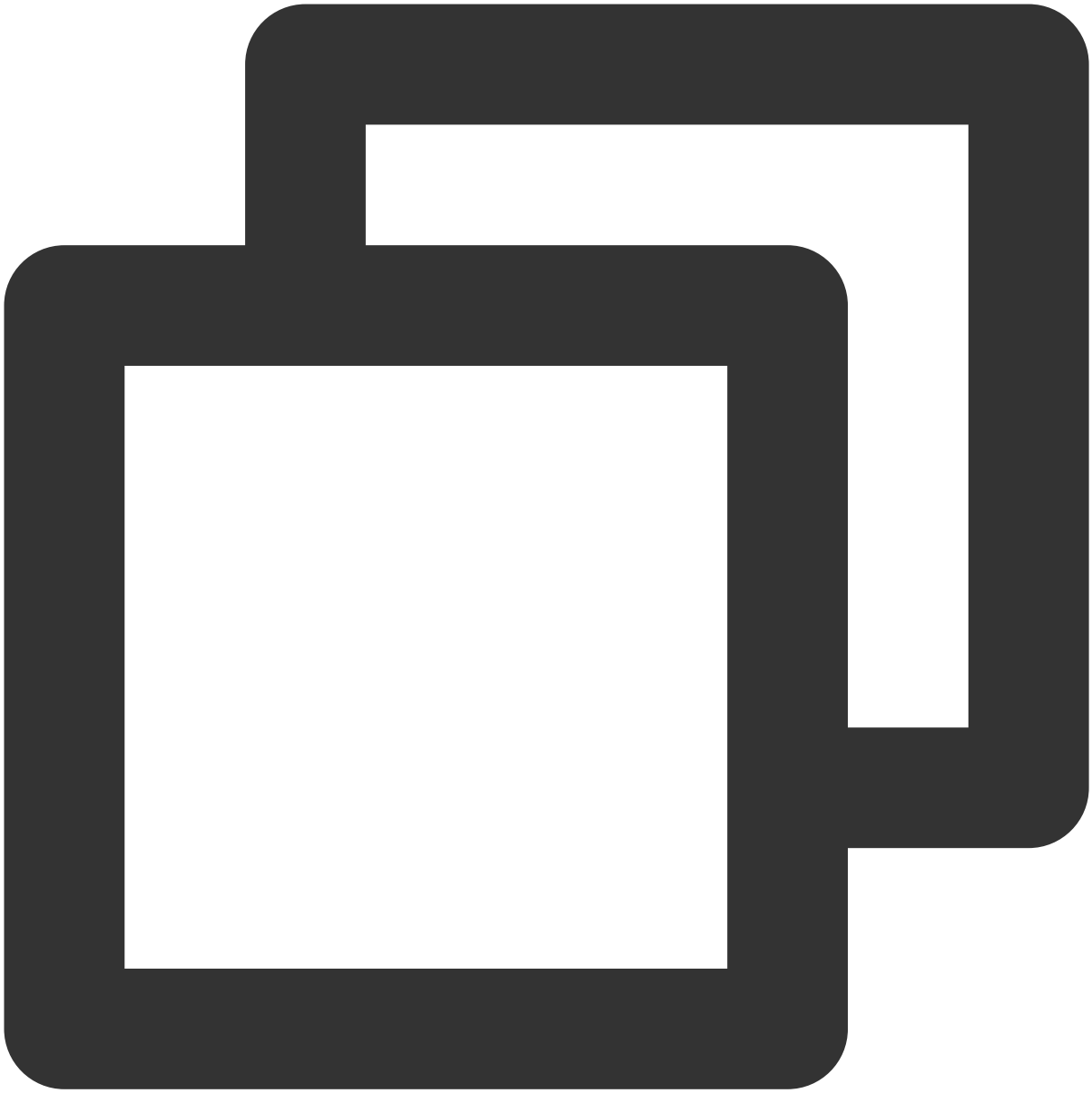
信息配置 > 资源名称填写 “dlc-demo”。新建资源详细介绍请参见购买独享数据引擎。



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}
```

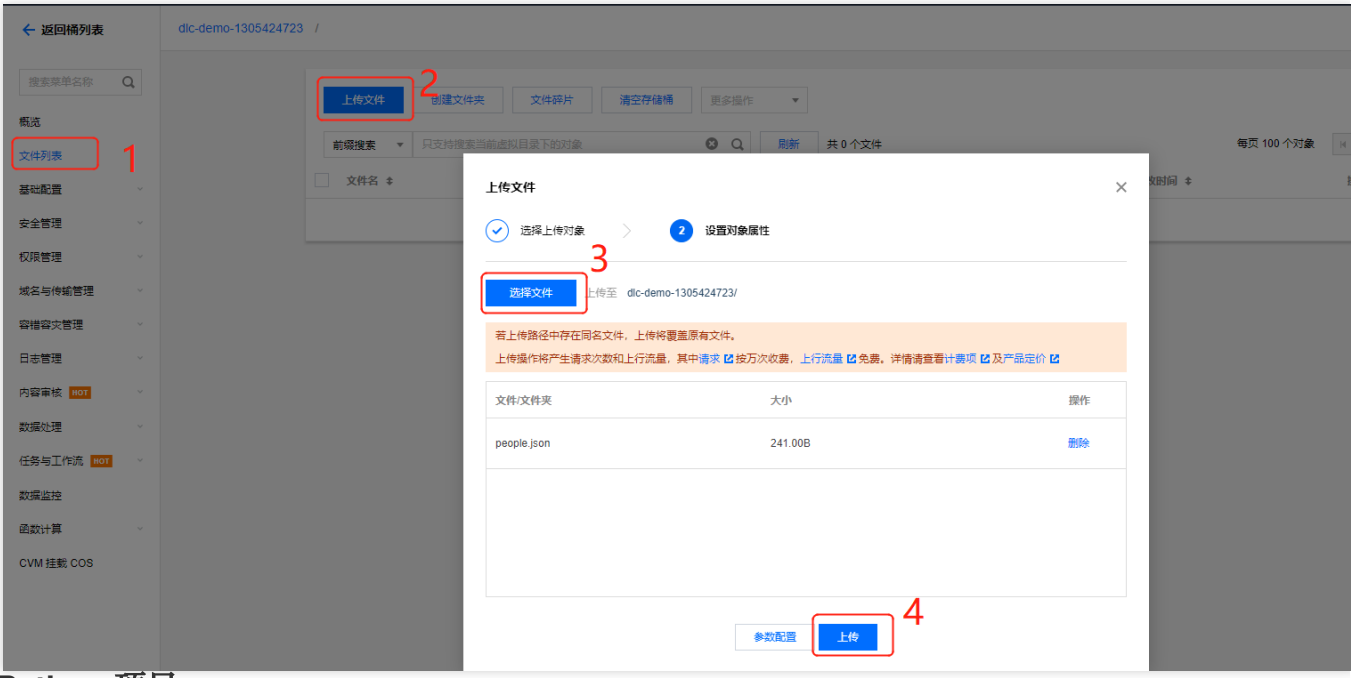
1. 登录 对象存储 COS 控制台，在左侧菜单导航中单击 **存储桶列表。**

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。

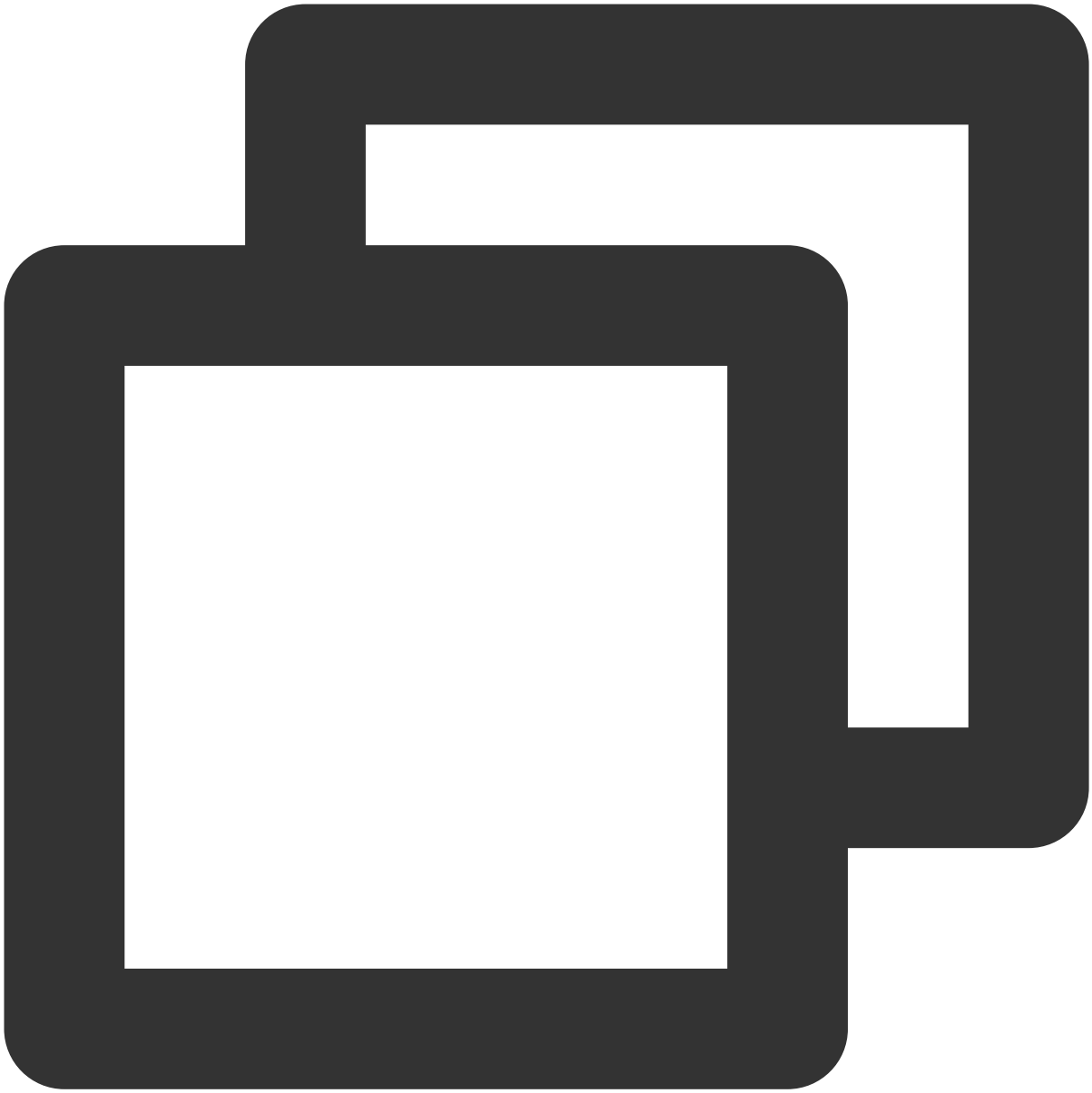


新建 Python 项目

通过 PyCharm 新建一个名称为“demo”的 项目。

编写代码

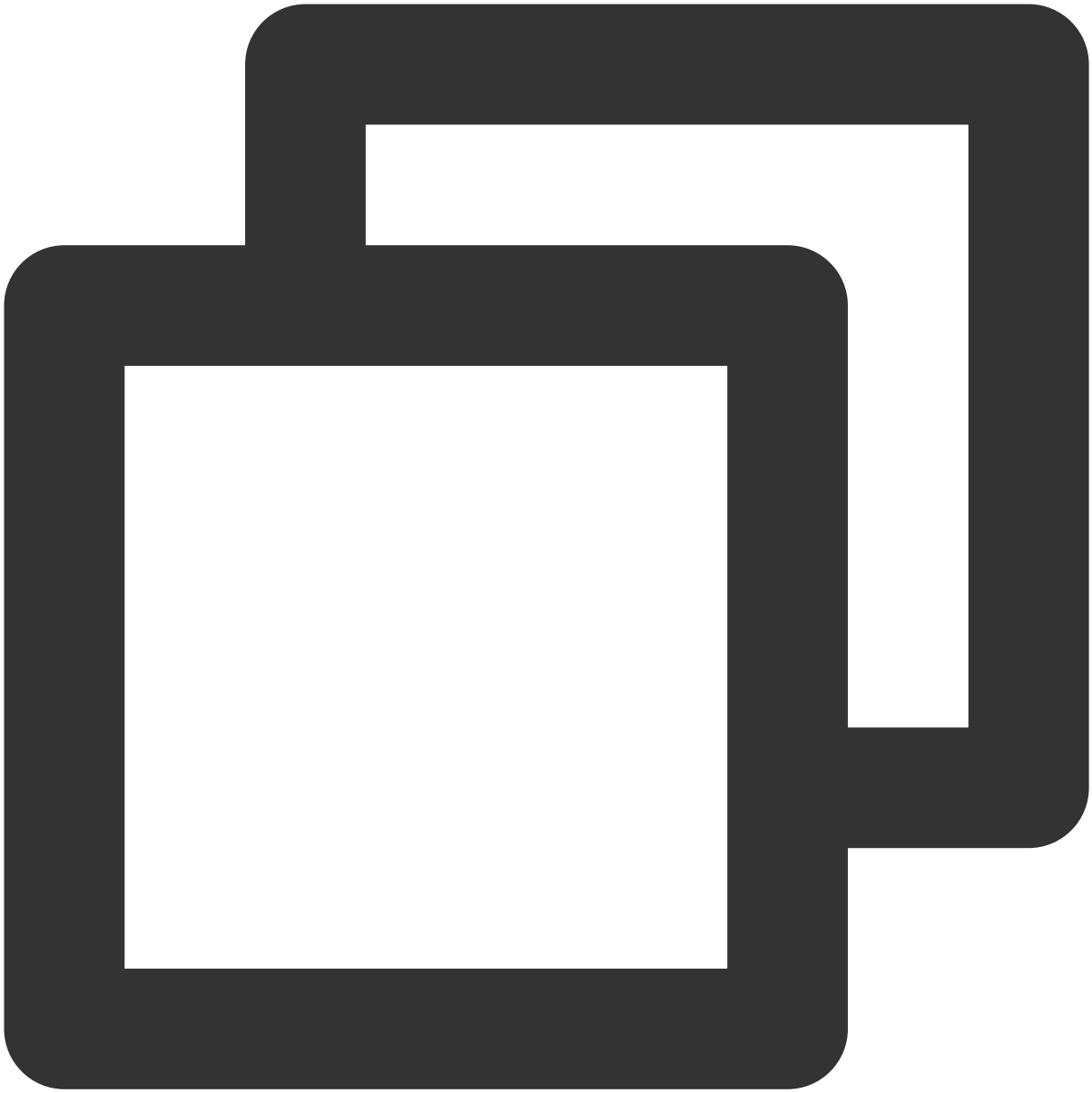
1. 新建 cos.py 文件，编写代码，功能为从 COS 上读写数据和在 DLC 上建库、建表、查询数据和写入数据。



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

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

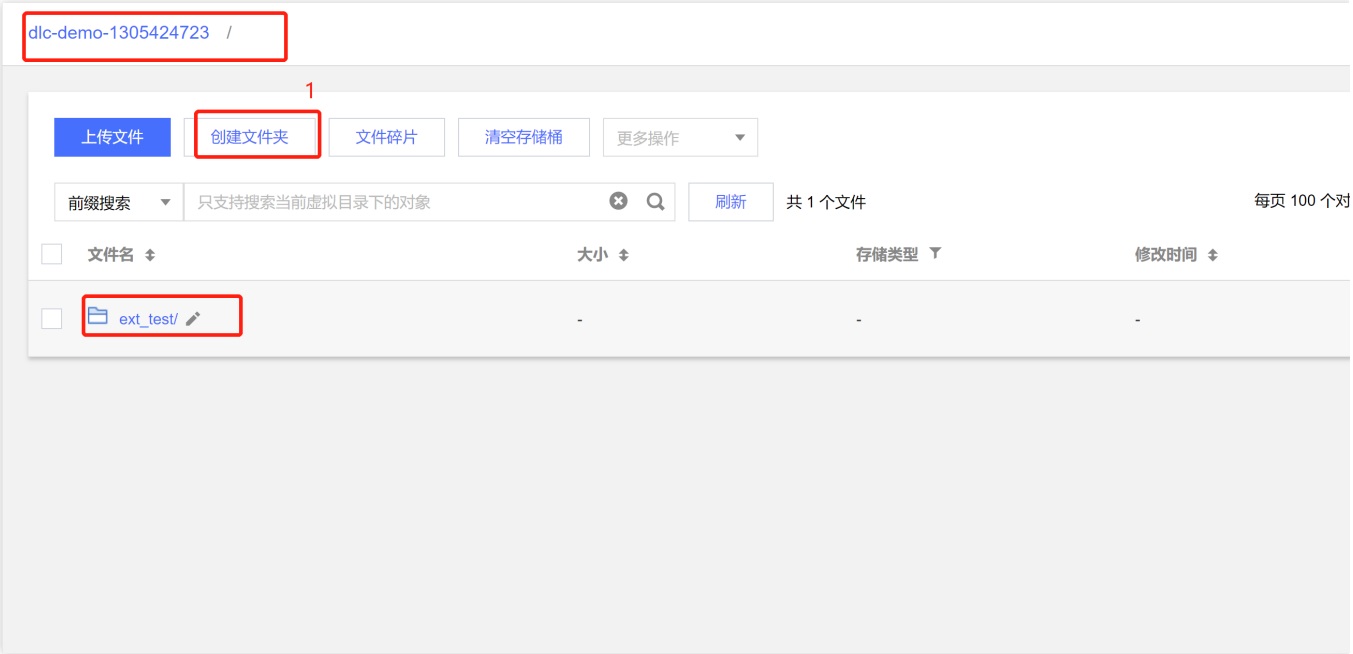
2. 新建 db.py 文件，编写代码，功能为 DLC 上建库、建表、查询数据和写入数据。



```
from os.path import abspath

from pyspark.sql import SparkSession



if __name__ == "__main__":

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

```
        .getOrCreate()

    # 1.建数据库
    spark.sql("CREATE DATABASE IF NOT EXISTS `DataLakeCatalog`.`dlc_db_test_py` COM
    # 2.建内表
    spark.sql("CREATE TABLE IF NOT EXISTS `DataLakeCatalog`.`dlc_db_test_py`.`test`
    # 3.写内数据
    spark.sql("INSERT INTO `DataLakeCatalog`.`dlc_db_test_py`.`test` VALUES (1,'And
    # 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_p
    # 6.写外数据
    spark.sql("INSERT INTO `DataLakeCatalog`.`dlc_db_test_py`.`ext_test` VALUES (1,
    # 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。

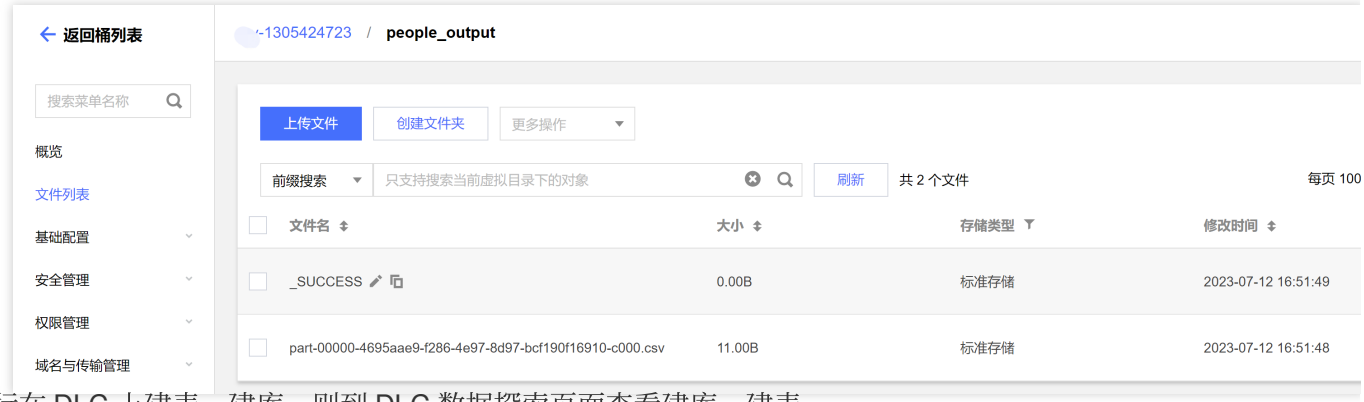
查看作业运行日志

1. 单击**作业名称 > 历史任务** 查看任务运行状态：

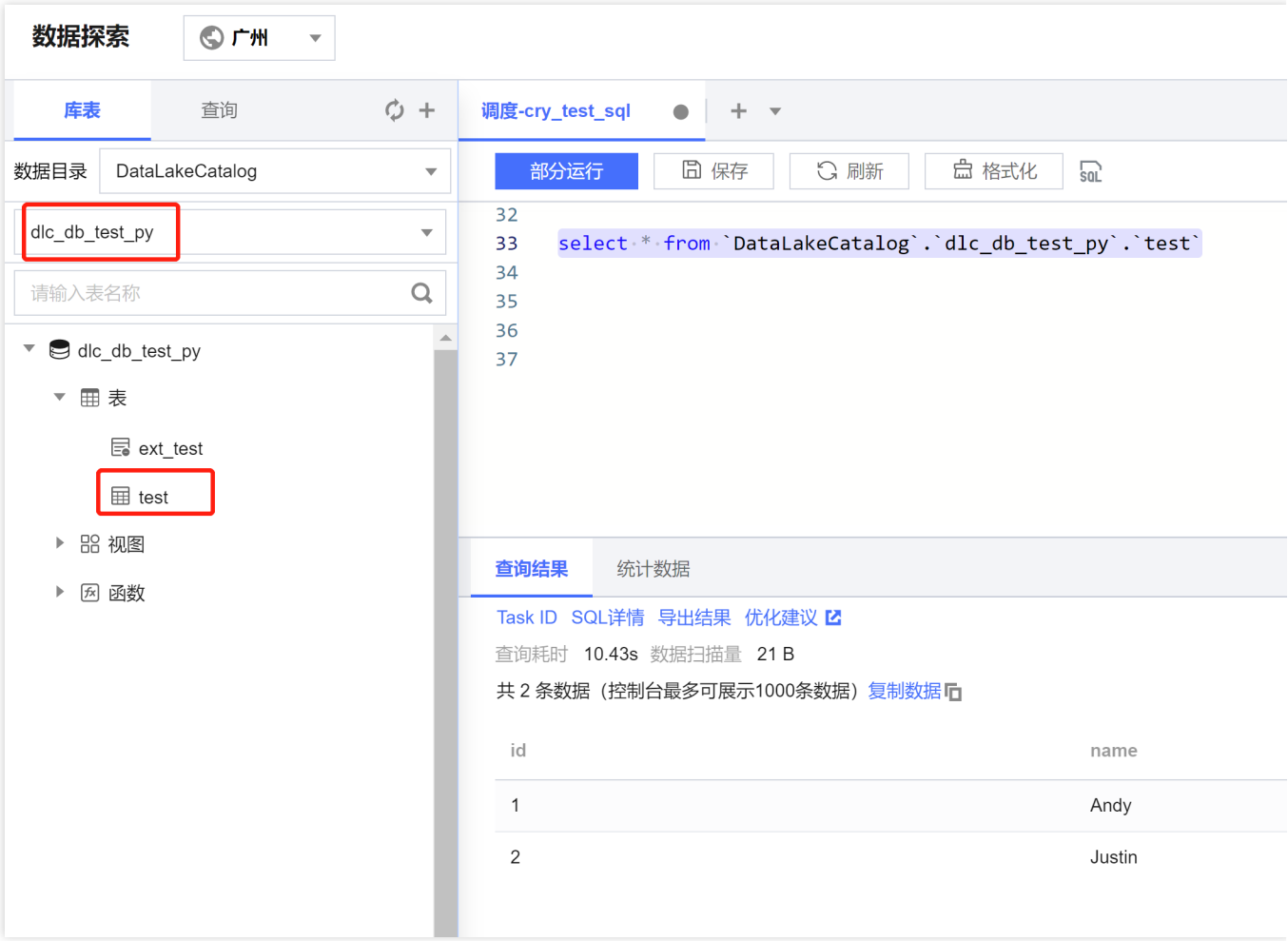
2. 单击**任务ID > 运行日志**，查看作业运行日志。

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：2024-07-31 18:03:13

前言

为了提升任务执行效率，DLC 引擎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。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。

优化 SQL 语句

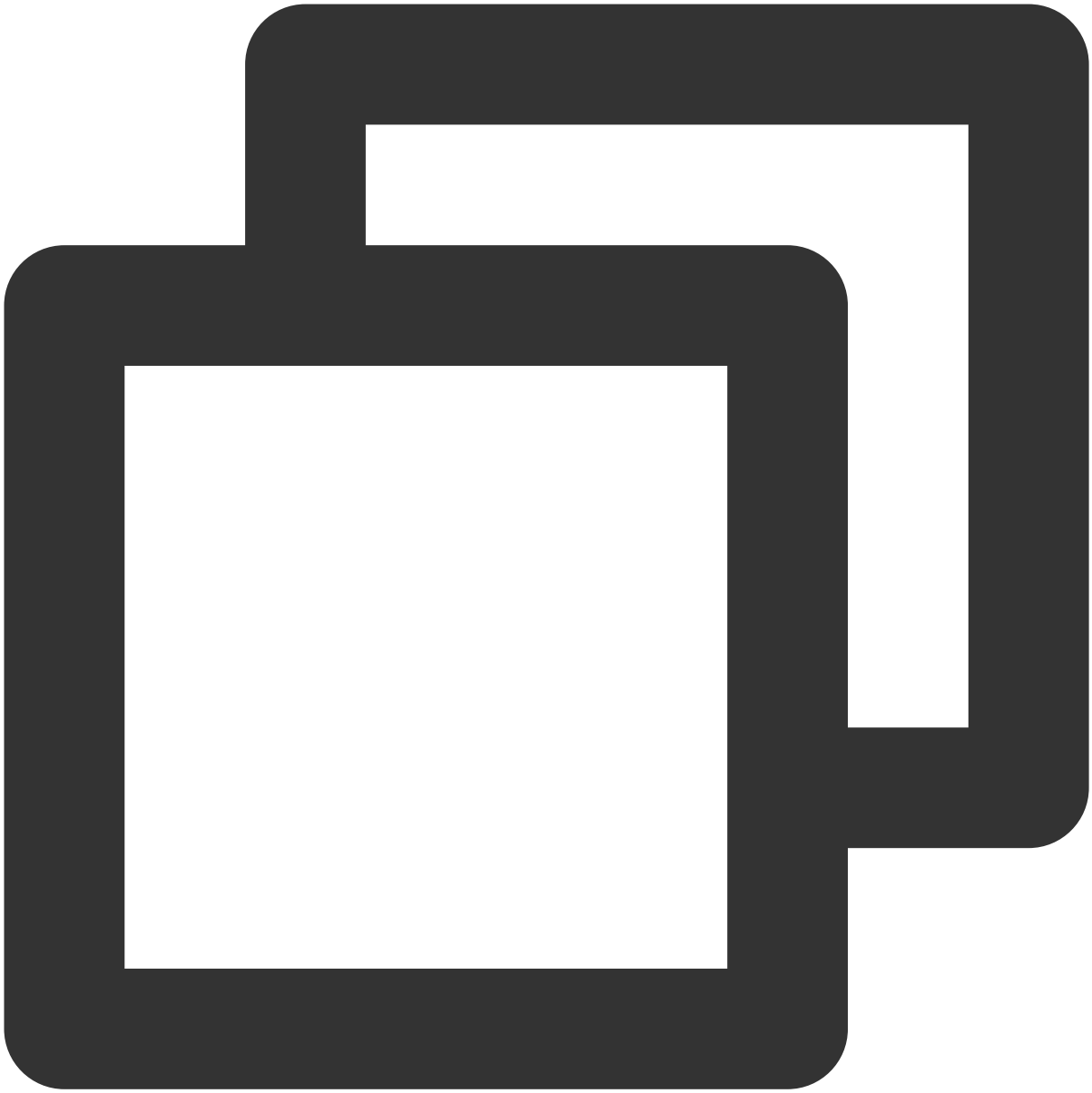
场景：SQL 语句本身不合理，导致执行效率不高。

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。

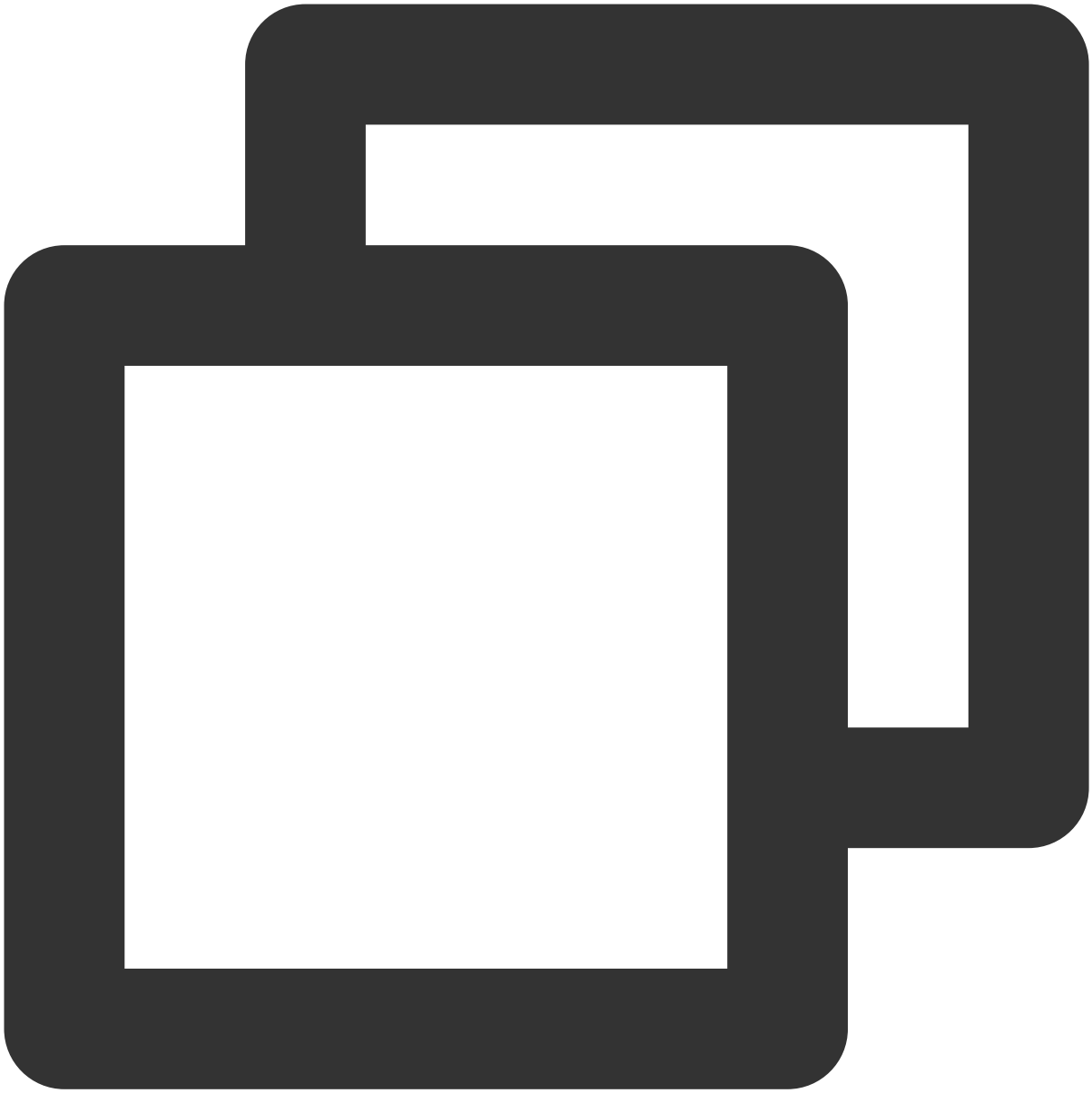
优化 GROUP BY 语句

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;
```

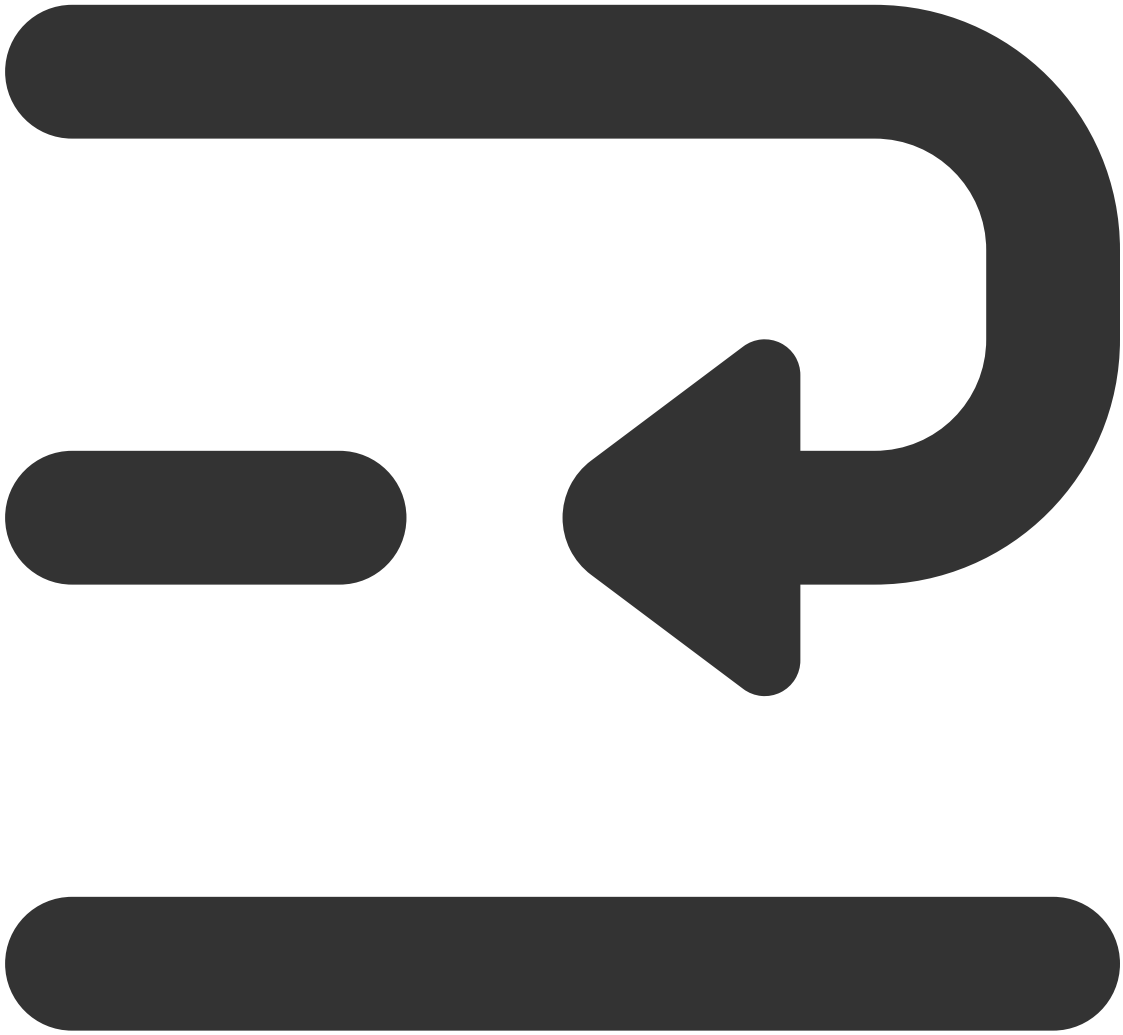
使用近似聚合函数

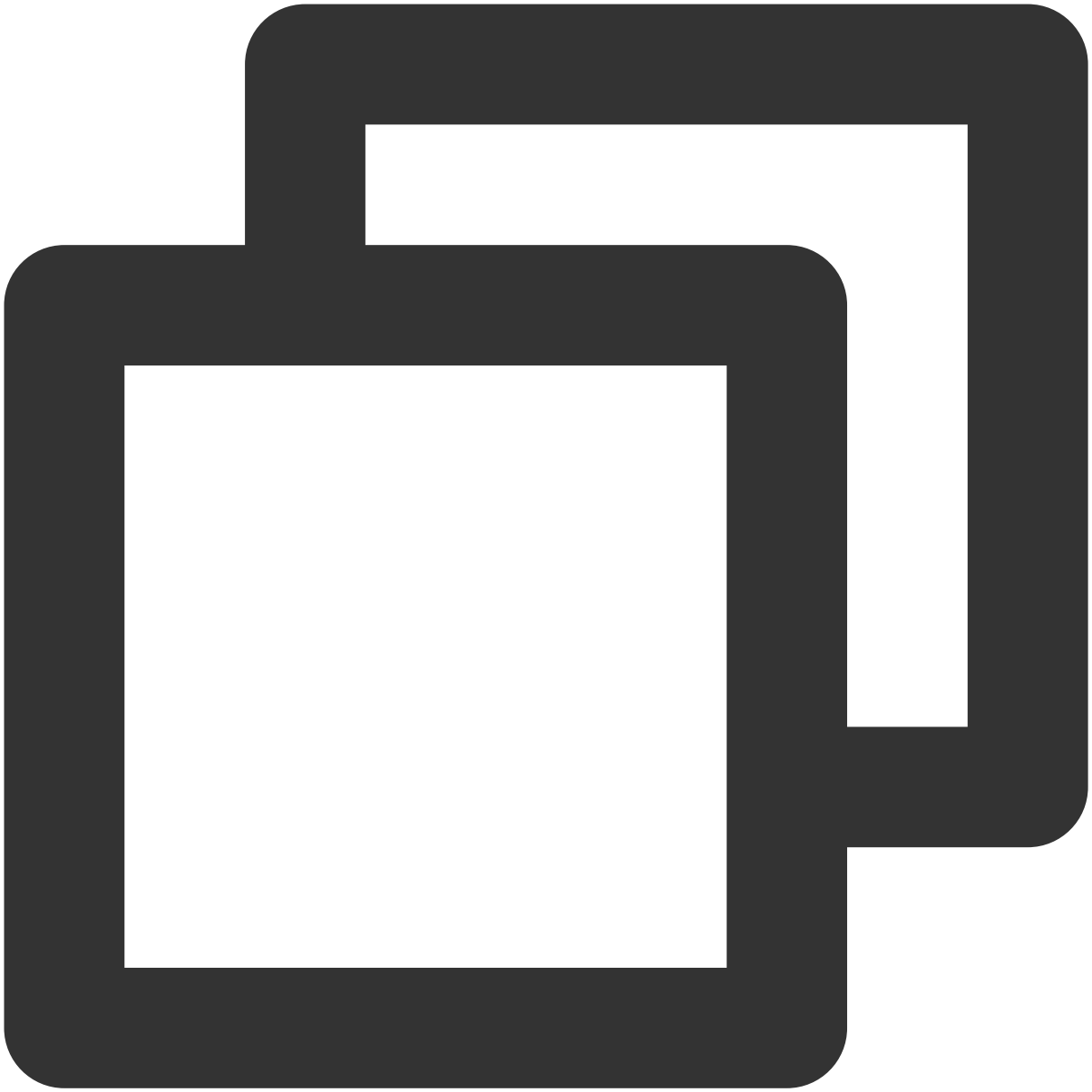
对于允许有少量误差的查询场景，使用这一些近似聚合函数对查询性能有大幅提升。

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

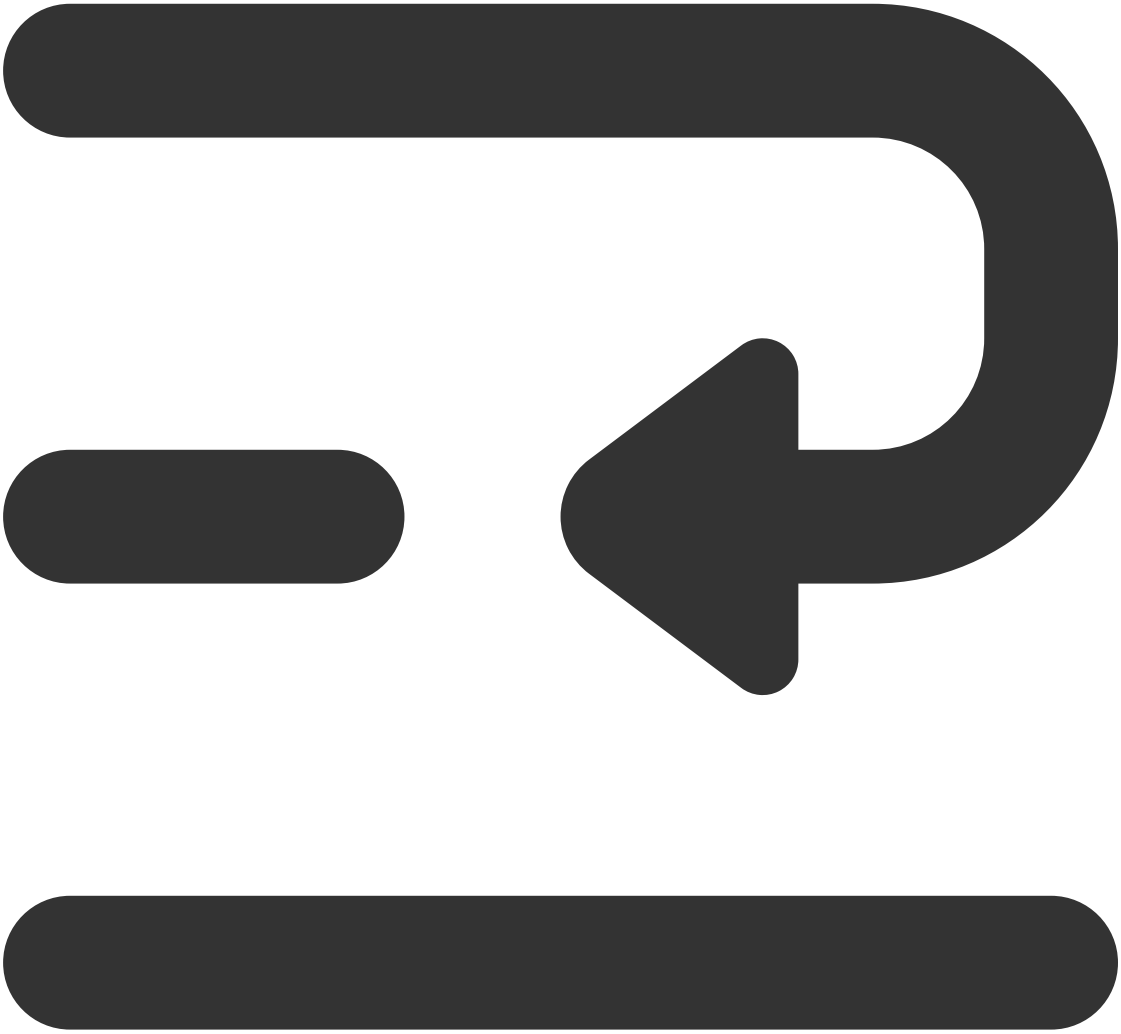
当 SQL 中有多个 LIKE 语句时，通常可以使用正则表达式来代替多个 LIKE，这样可以大幅提升执行效率。例如：

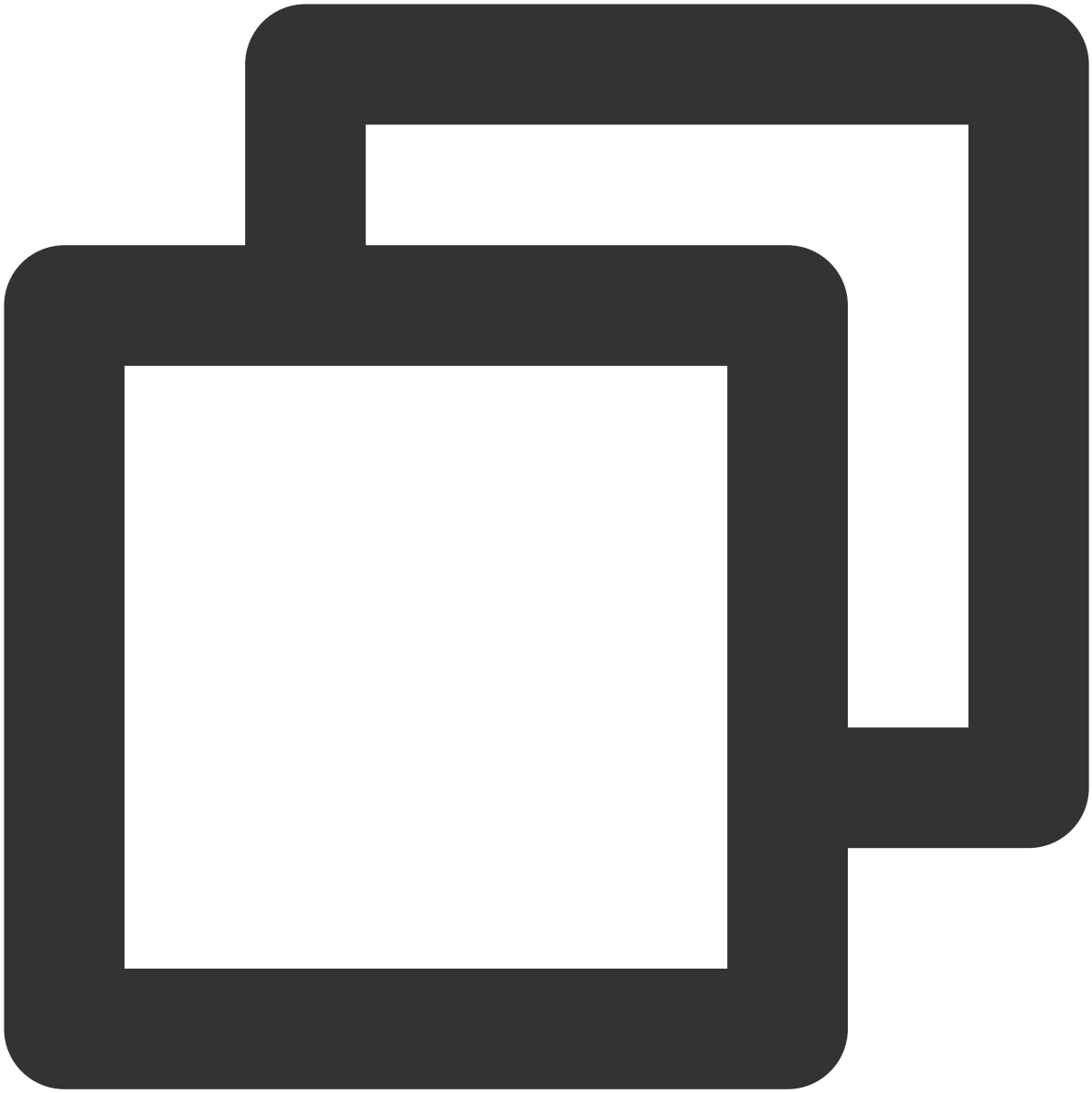




```
SELECT COUNT(*) FROM table_name WHERE field_name LIKE '%guangzhou%' OR LIKE '%beiji
```

可以优化成：





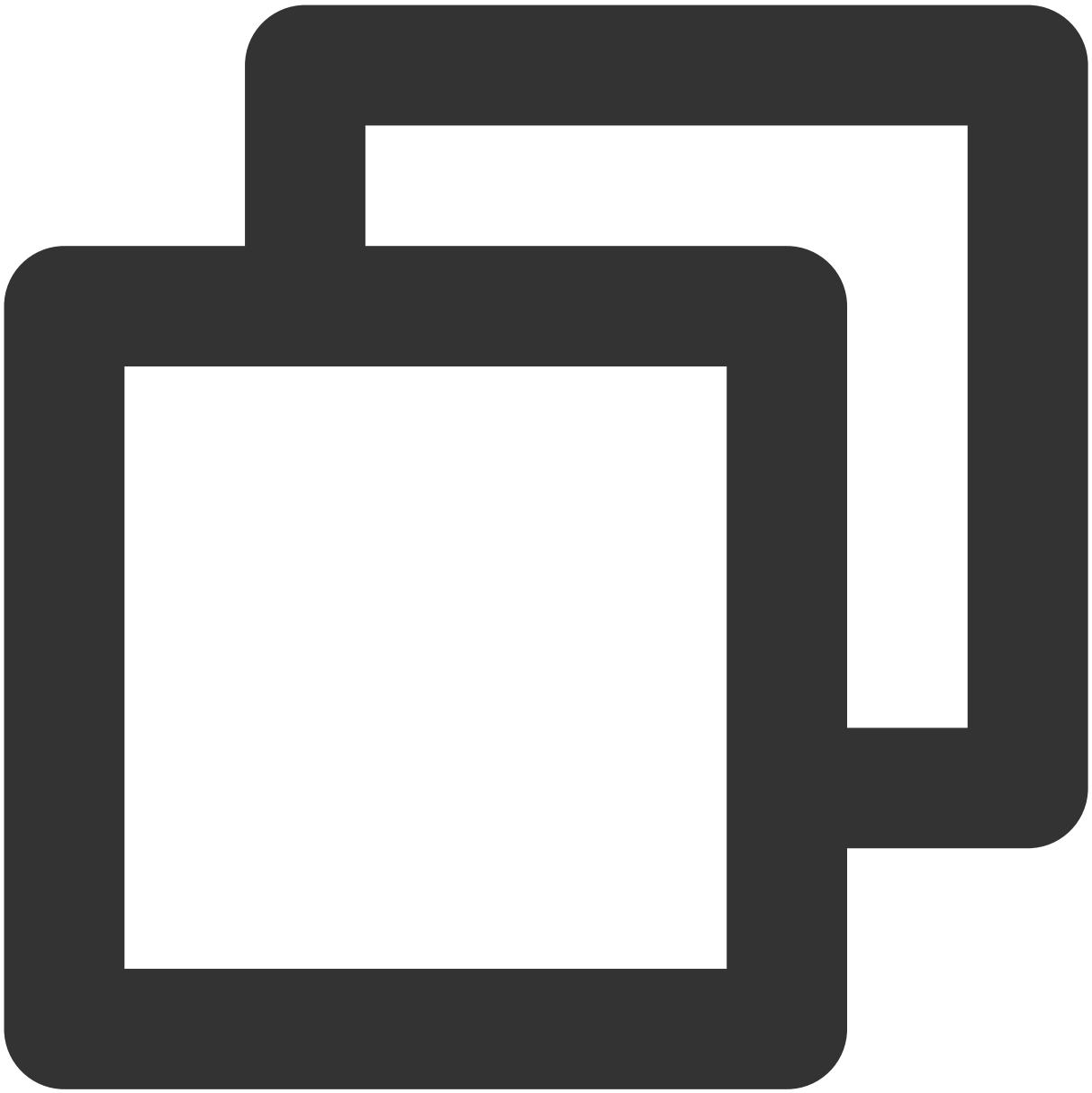
```
SELECT COUNT(*) FROM table_name WHERE regexp_like(field_name, 'guangzhou|beijing|ch
```

数据治理

数据治理适用场景

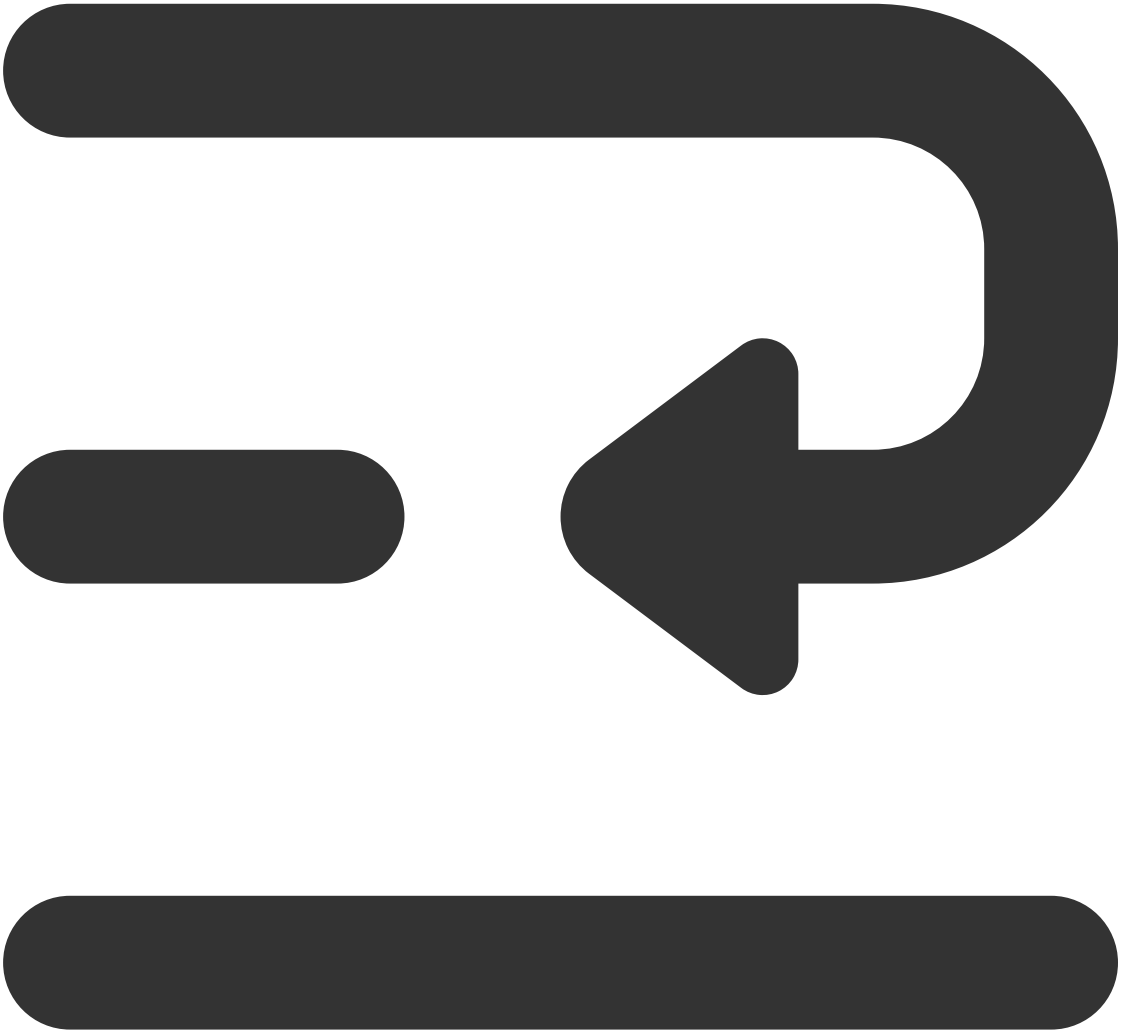
场景：实时写入。Flink CDC 实时写入通常采用 upsert 的方式写入，该流程在写入过程中会产生大量的小文件，当小文件堆积到一定程度后会导致数据查询变慢，甚至超时无法查询。

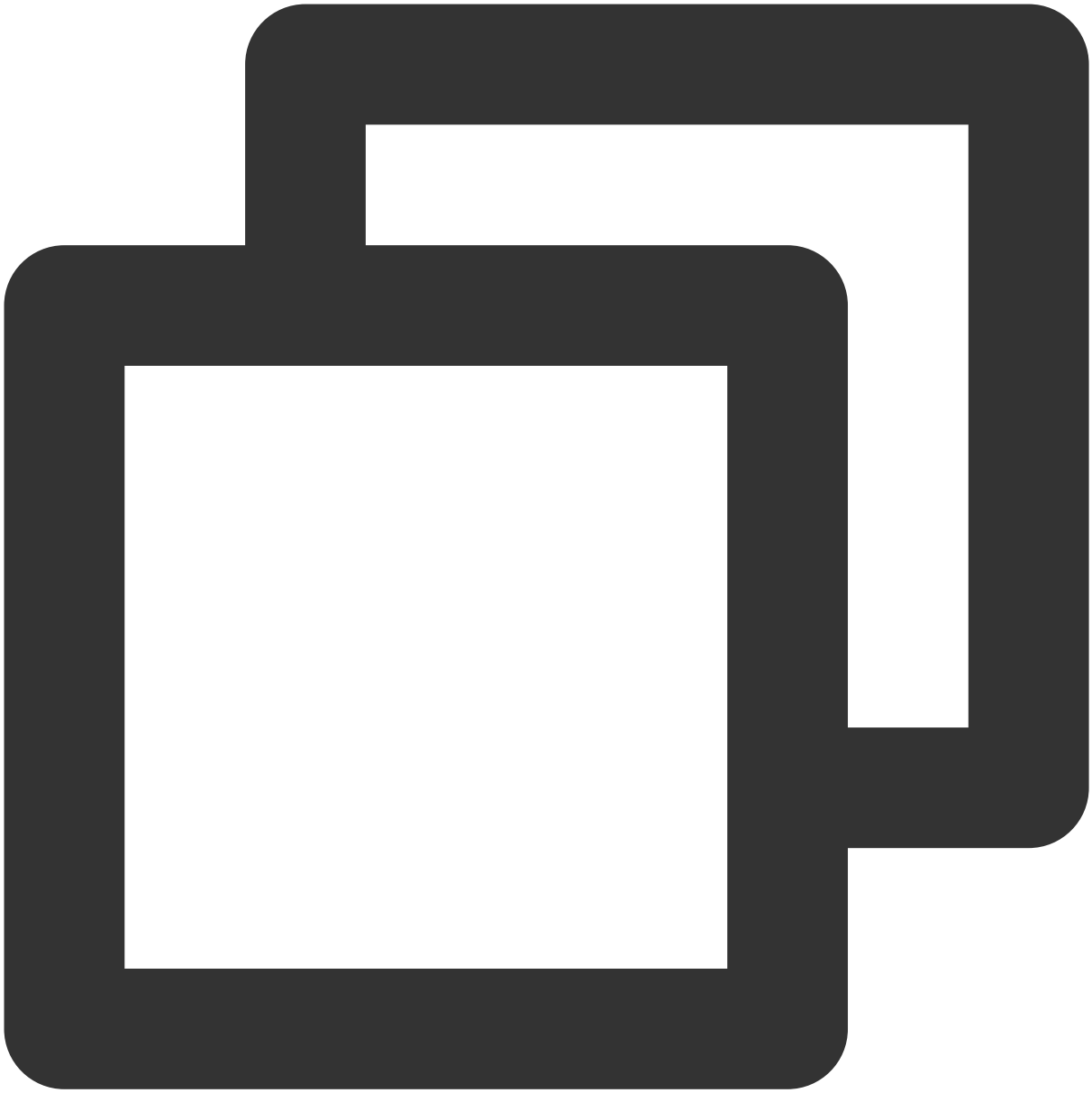
可以通过以下方式查看表文件数量和快照信息。



```
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 开启数据优化 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'	2.6s	235.9MB	480ms	16.5 KB	快81%	少99.9%
SELECT count(*) FROM tb WHERE dt<'2022-01-08' AND dt>'2001-07-08'	3.8s	401.6MB	2.2s	2.8MB	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。

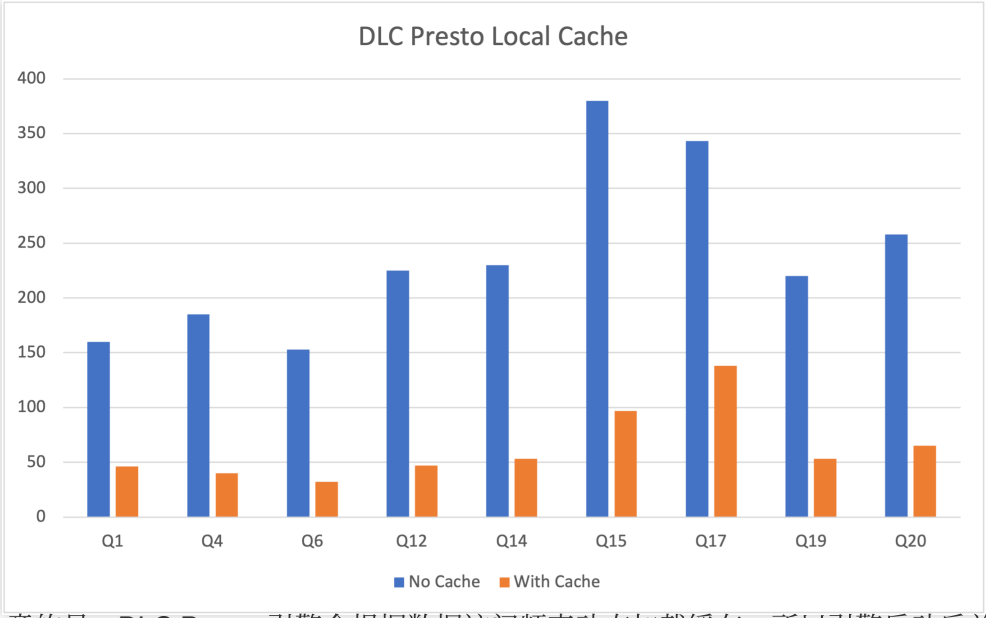
Alluxio ：是一种数据编排技术 。它提供缓存，将数据从存储层移动到距离数据驱动型应用更近的位置从而能够更容易被访问。Alluxio 内存至上的层次化架构使得数据的访问速度能比现有方案快几个数量级。

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。

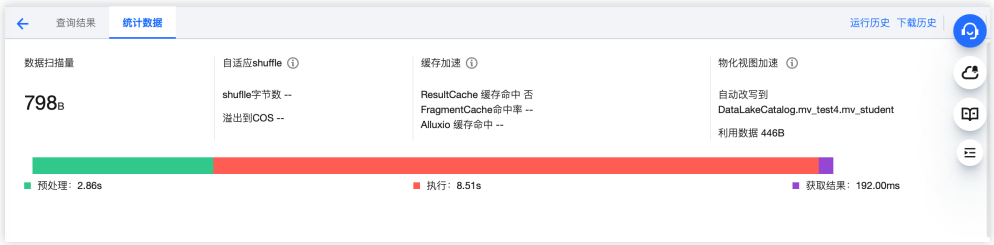
下表是在总数据量 为1TB的 Parquet 文件中的 TPCH 测试数据，本次测试选用16CU Presto。因为测试的是缓存功能，所以主要从 TPCH 中选择 IO 占用比较大的 SQL ，涉及的表主要有 lineitem、orders、customer 等表，涉及的 SQL 为 Q1、Q4、Q6、Q12、Q14、Q15、Q17、Q19 以及 Q20。其中横坐标表示SQL语句，纵坐标表示运行时间(单位秒)。



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

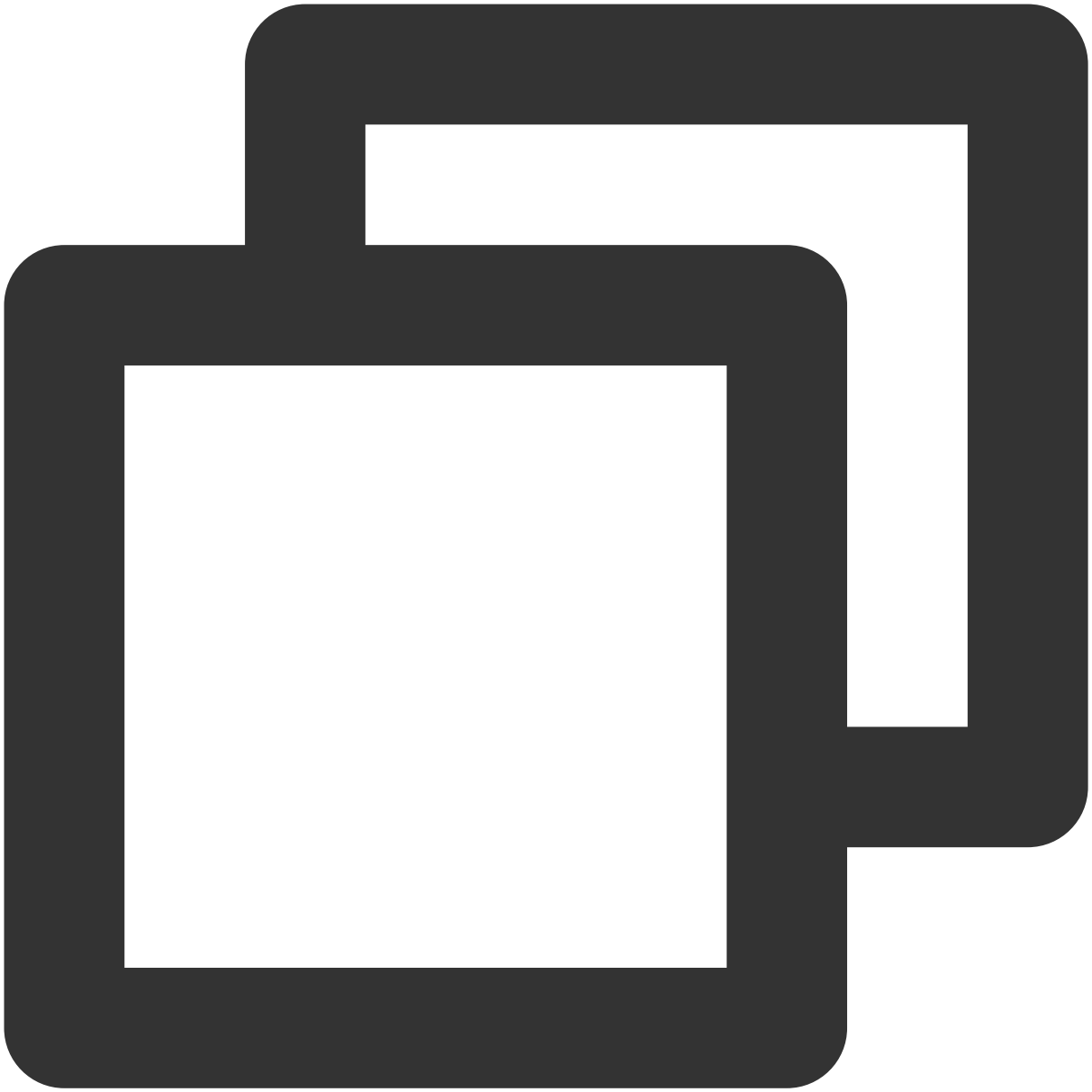
您可以在DLC控制台 数据探索 功能中查看执行的SQL任务的缓存命中情况。



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

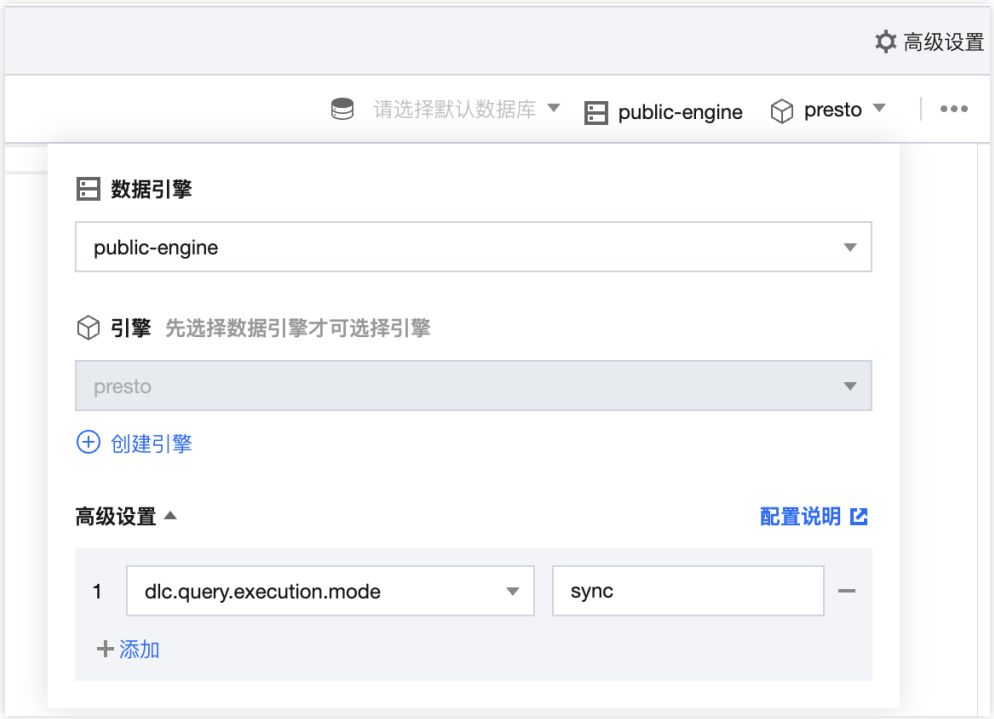
表类型	查询	耗时	数据扫描量
外表	第一次查询	16.5s	2.42GB
	第二次查询	15.3s	2.42GB
	第三次查询	14.3s	2.42GB

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

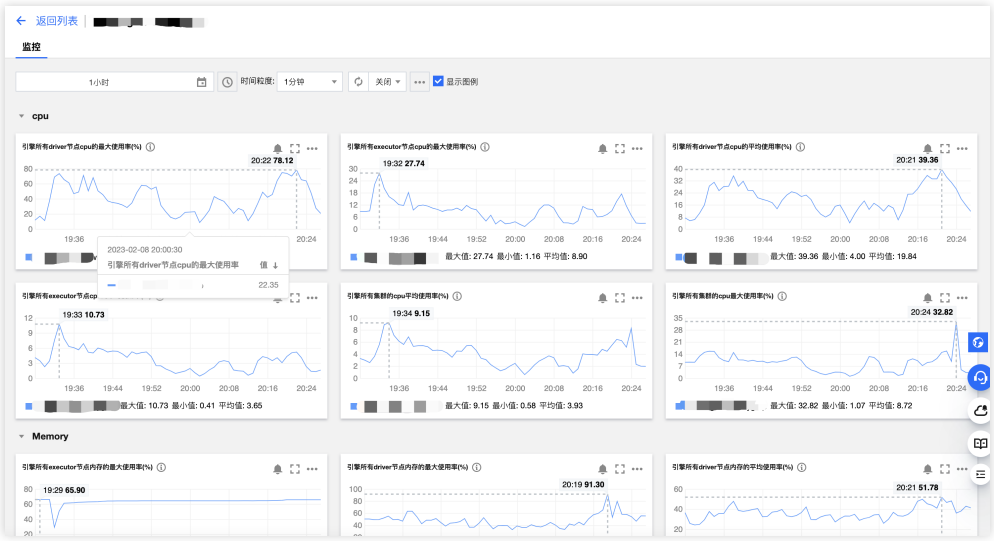
DLC 针对于 BI 场景进行了特别的优化，可以通过配置引擎参数dlc.query.execution.mode来开启同步模式或者异步模式（只支持 presto 引擎）。取值介绍如下。
async（默认）：该模式任务会完成全量查询计算，并将结果保存到 COS，再返回给用户，允许用户在查询完成后下载查询结果。
sync：该模式下，查询不一定会执行全量计算，部分结果可用后，会直接由引擎返回给用户，不再保存到 COS。因此用户可获得更低查询延迟和耗时， 但结果只在系统中保存30s。推荐不需要从 COS 下载完整查询结果，但期望更低查询延迟和耗时时使用该模式，例如查询探索阶段、BI 结果展示。
配置方式：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

为了提高稳定性，DLC 默认开启自适应 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。

集群冷启动

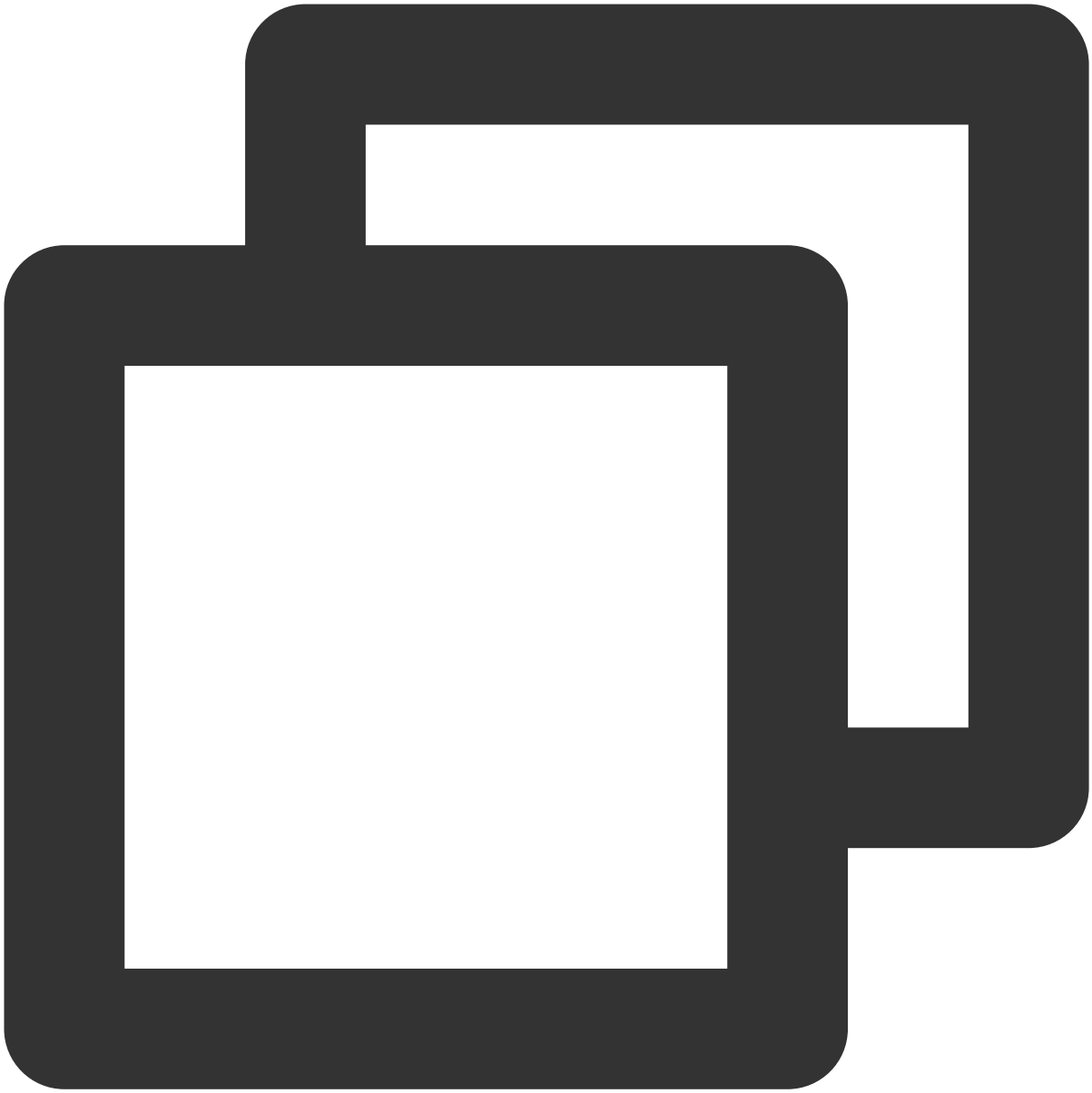
DLC 支持自动或者手动挂起集群，挂起后不再产生费用，所以在集群启动后，首次执行任务可能存在“正在排队”的提示，这是因为集群冷启动中正在拉起资源。如果您频繁提交任务，建议购买包年包月集群，该类型集群不存在冷启动，能在任何时间快速执行任务。

UDF 函数开发指南

最近更新时间：2024-07-31 18:03:25

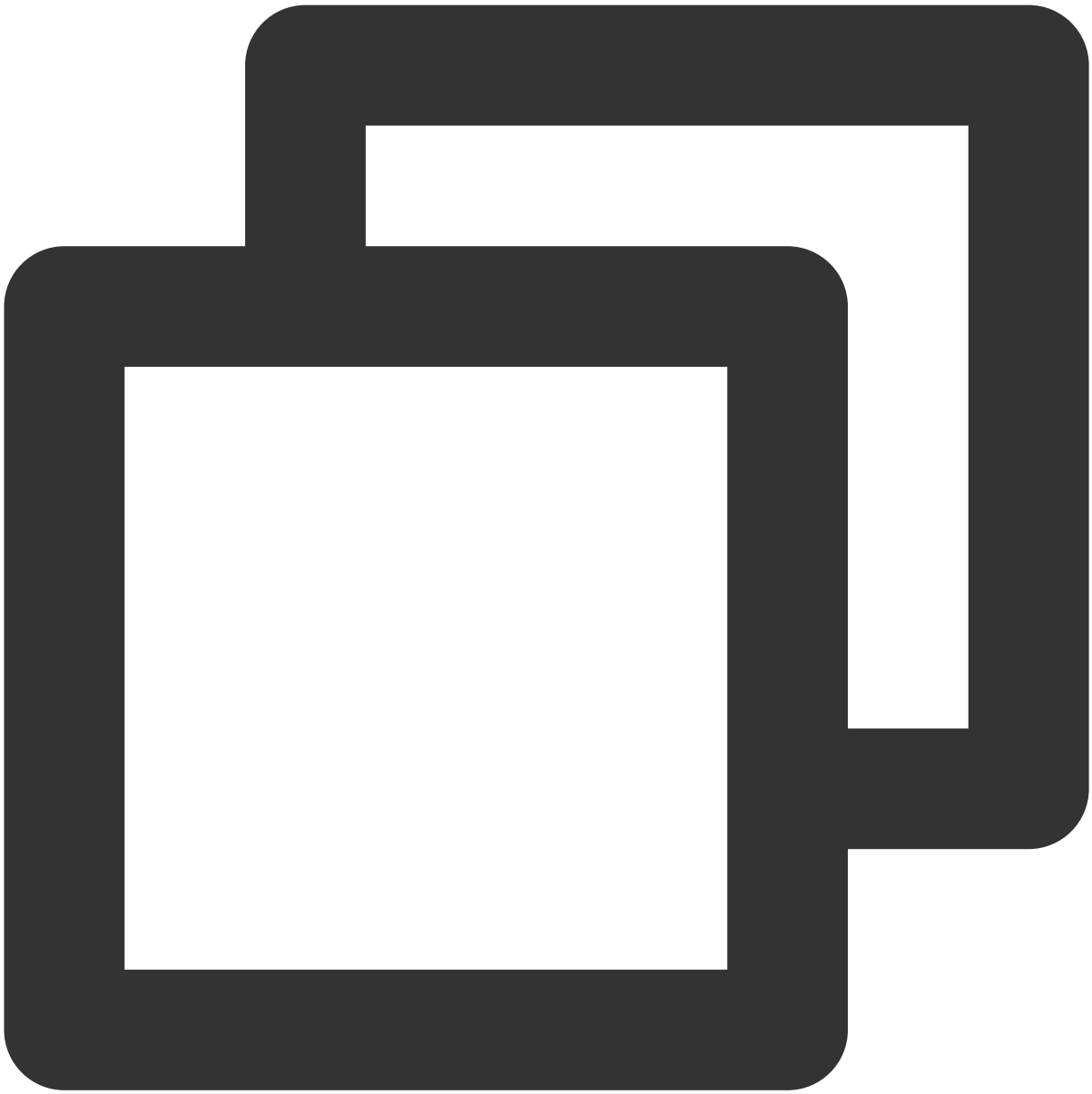
UDF 说明

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

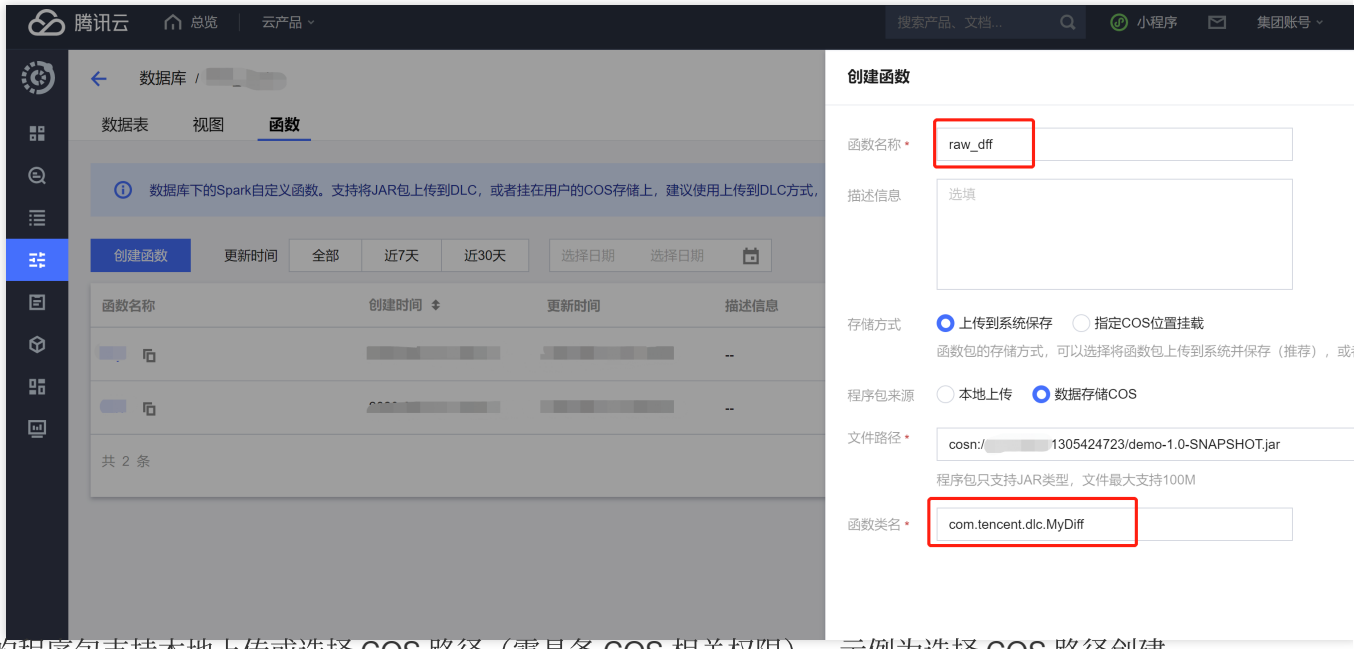
注意：

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 数据目录及数据库管理。

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。


运行
保存
刷新
格式化
SQL
35 select raw_diff(Array[1,2,3])
36
37
查询结果
统计数据
Task ID SQL详情 导出结果 优化建议
查询耗时 5.21s
共 1 条数据（控制台最多可展示1000条数据）复制数据
test_cache.raw_diff(array(1, 2, 3))
[0,1,1]

物化视图

最近更新时间：2024-07-31 18:03:40

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。

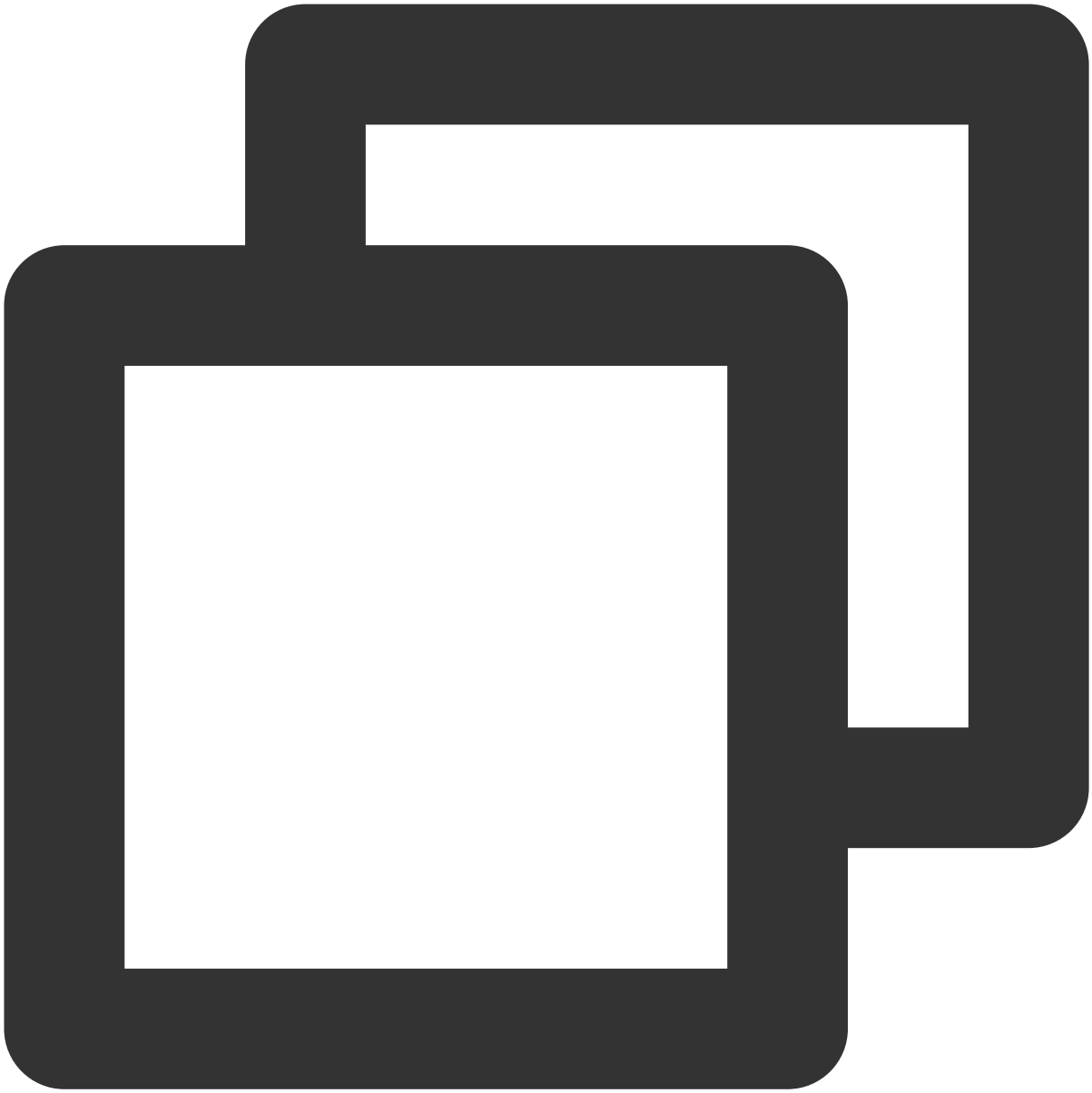
普通物化视图

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。

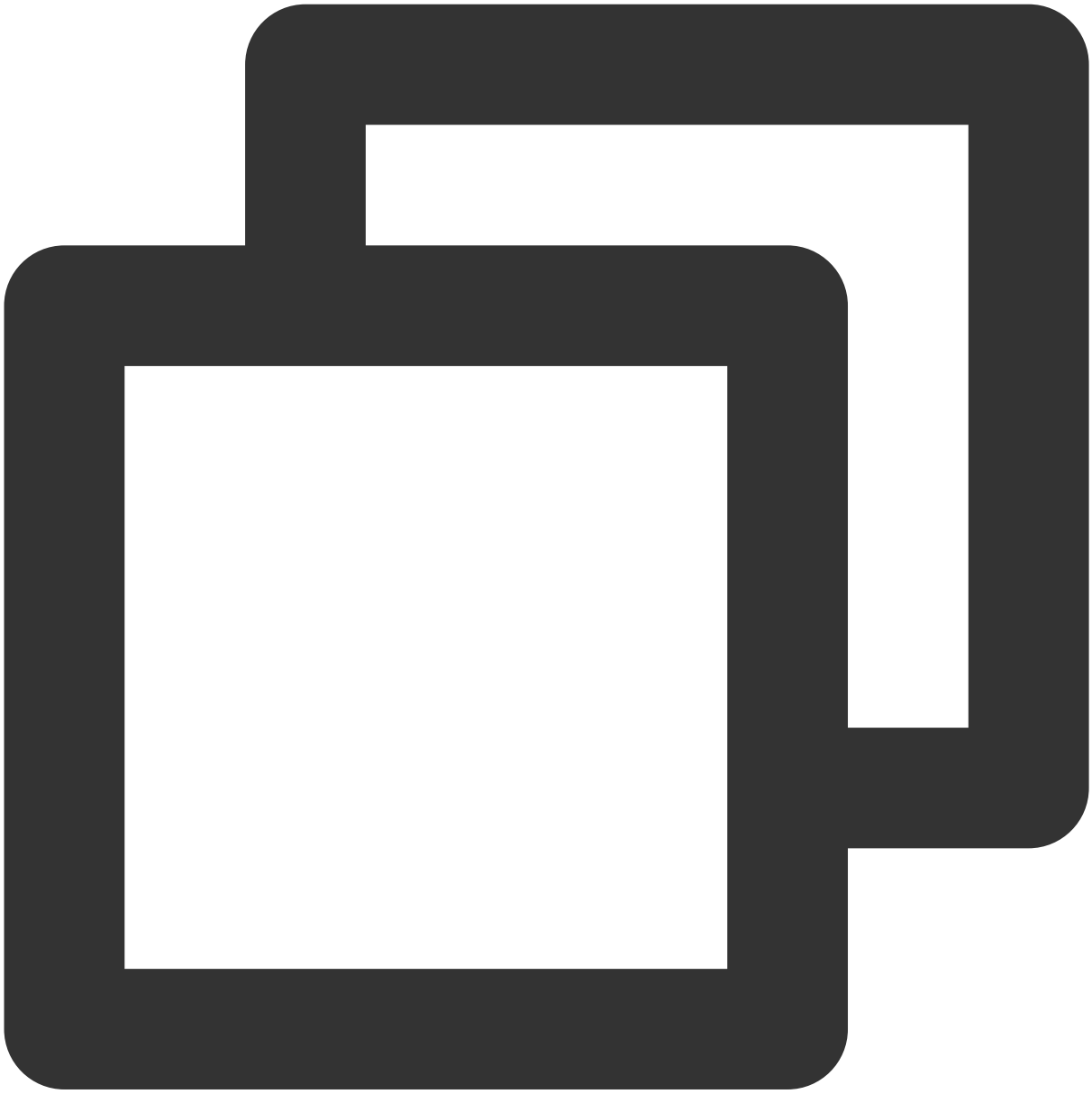


```
CREATE DATABASE IF NOT EXISTS mv_test3;
create table student(id int, name string, score int);
insert into student values (1,'zhangsan', 90);
insert into student values (2,'lisi', 100);
insert into student values (3,'wangwu', 80);
insert into student values (4,'zhaoliu', 30);
select * from student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

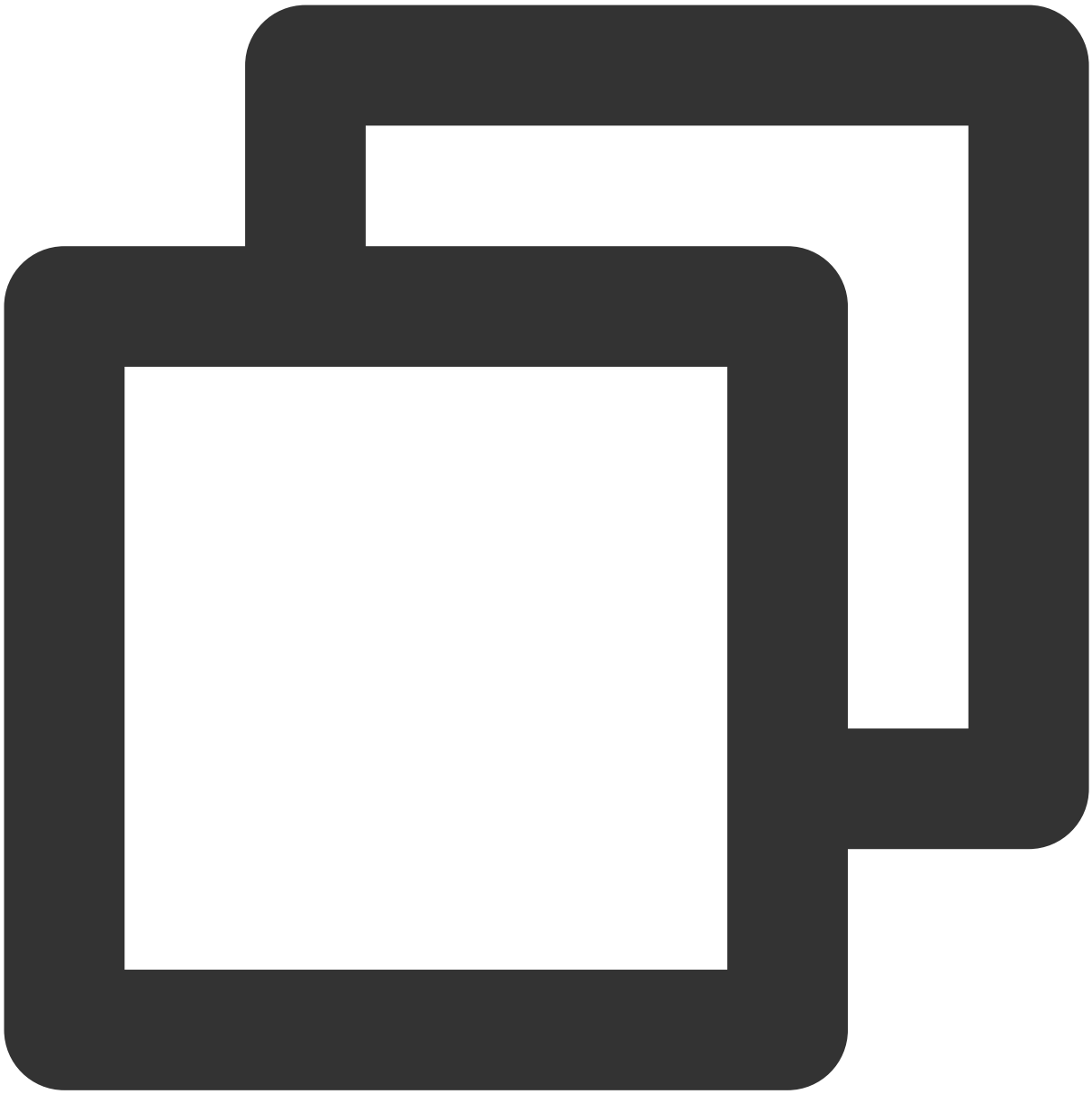
以下例子，使用了一个简单的 SELECT 语句从表 `student` 中选择所有分数，并对它们进行求和操作。然后将这个求和结果作为物化视图 `mv_student_sum` 的内容。



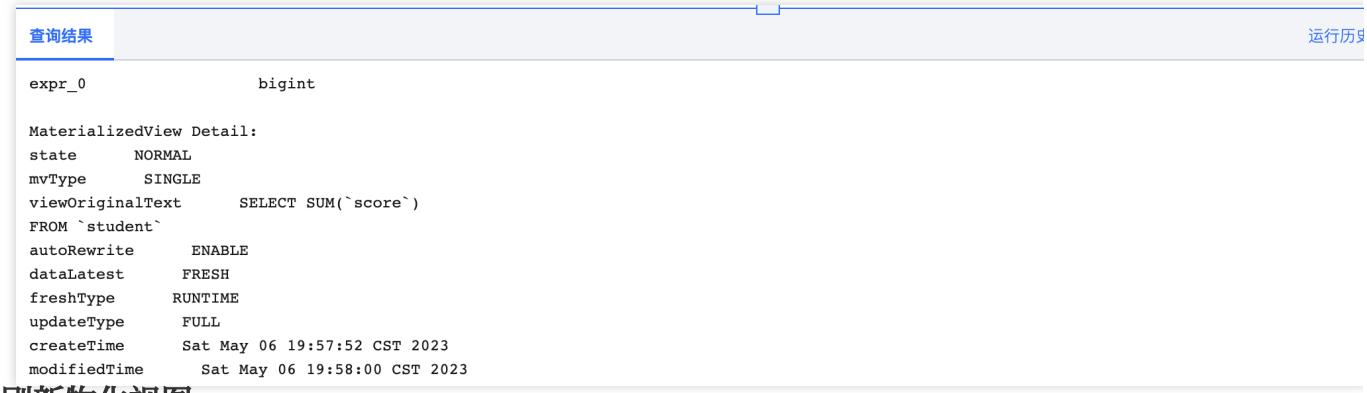
```
CREATE MATERIALIZED VIEW mv_student_sum AS (
  select sum(score) from student
);
```

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。



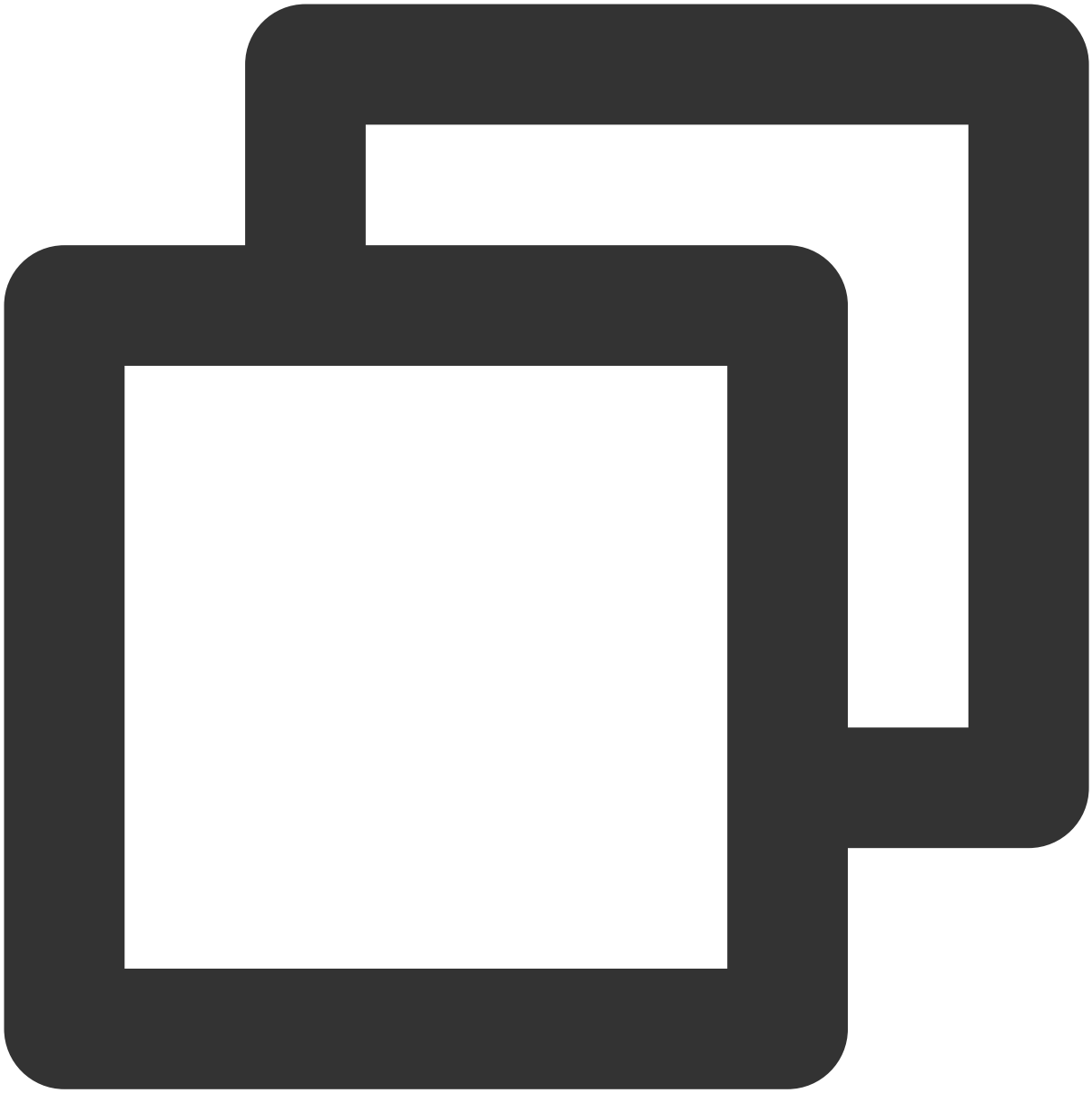
```
DESCRIBE MATERIALIZED VIEW mv_student_sum;
```



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。

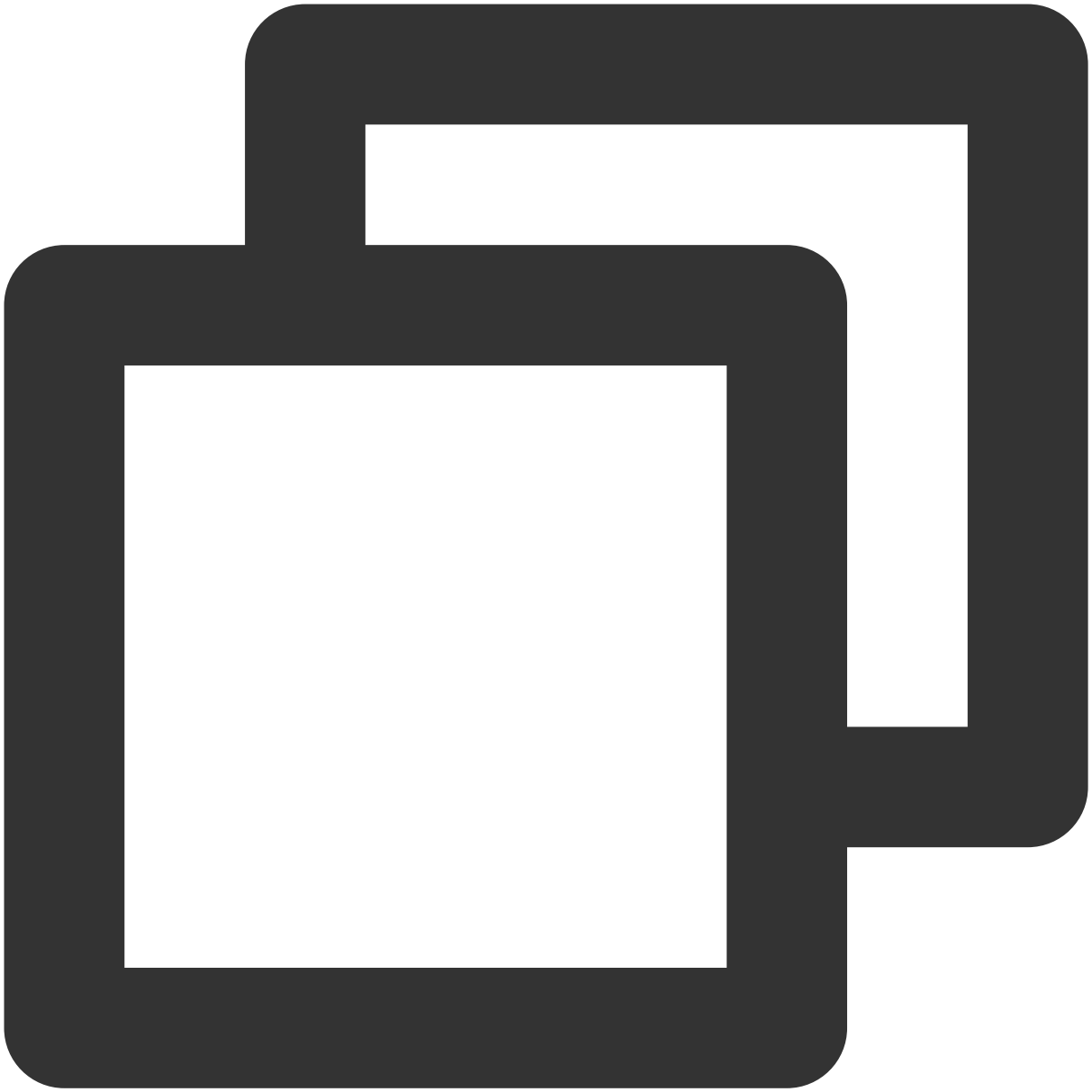
此处仅作演示，大部分情况下，您并不需要手动刷新物化视图，只要 SQL 命中了源表有变更的物化视图就会自动刷新。



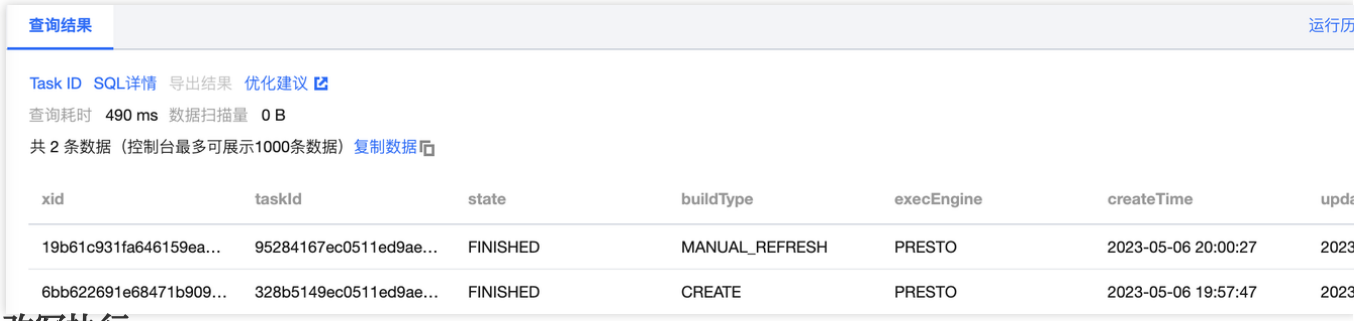
```
REFRESH MATERIALIZED VIEW mv_student_sum;
```

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。

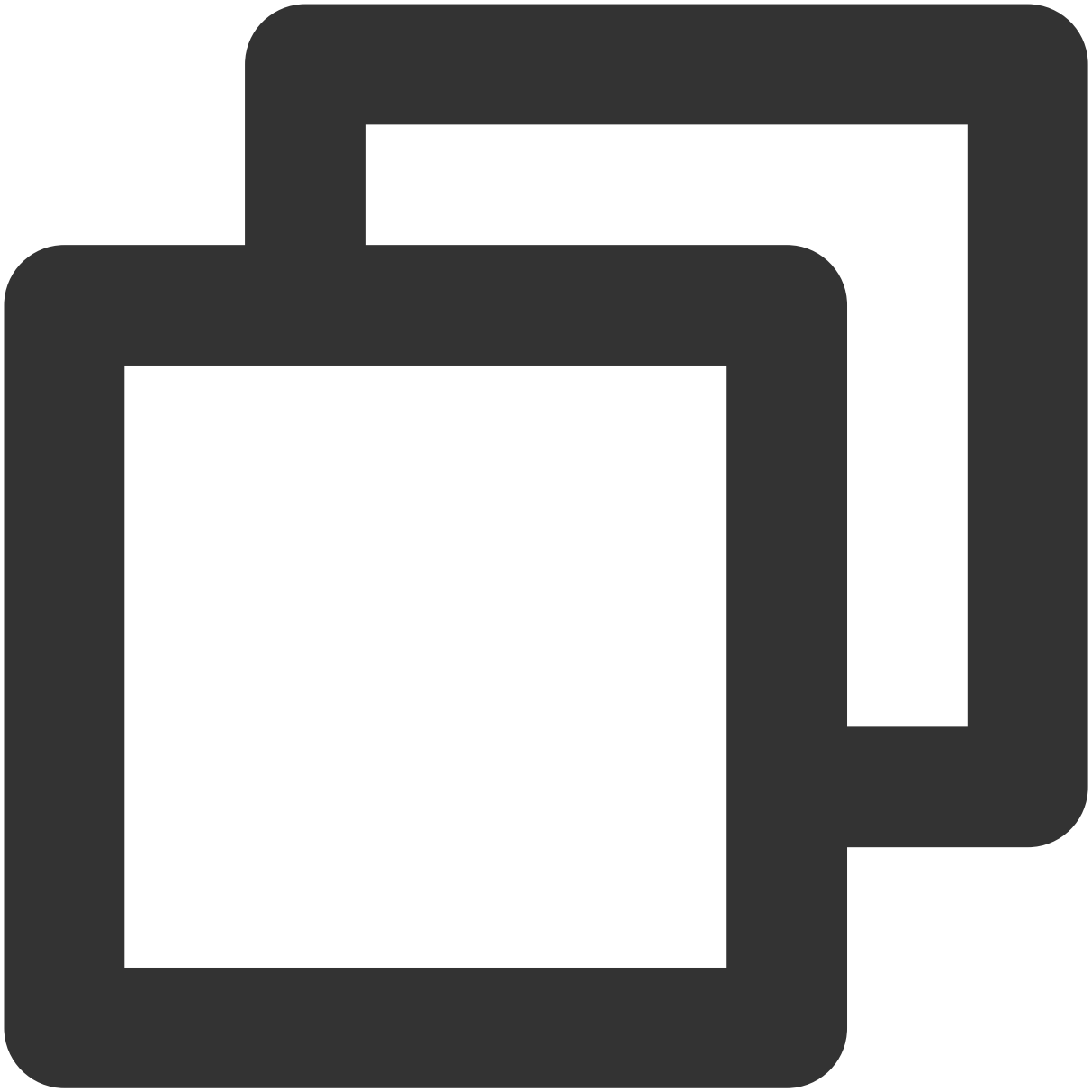


```
SHOW MATERIALIZED VIEW JOBS IN mv_student_sum;
```

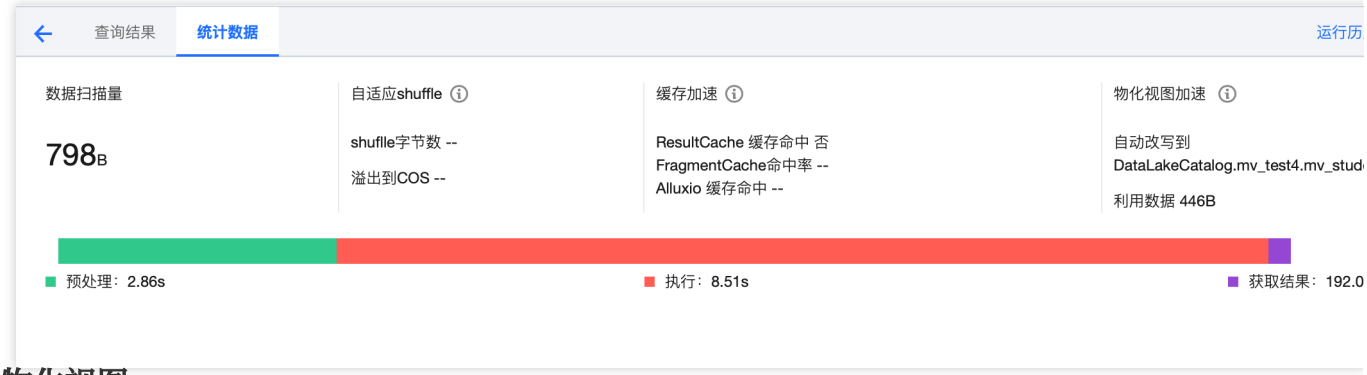


SQL 改写执行

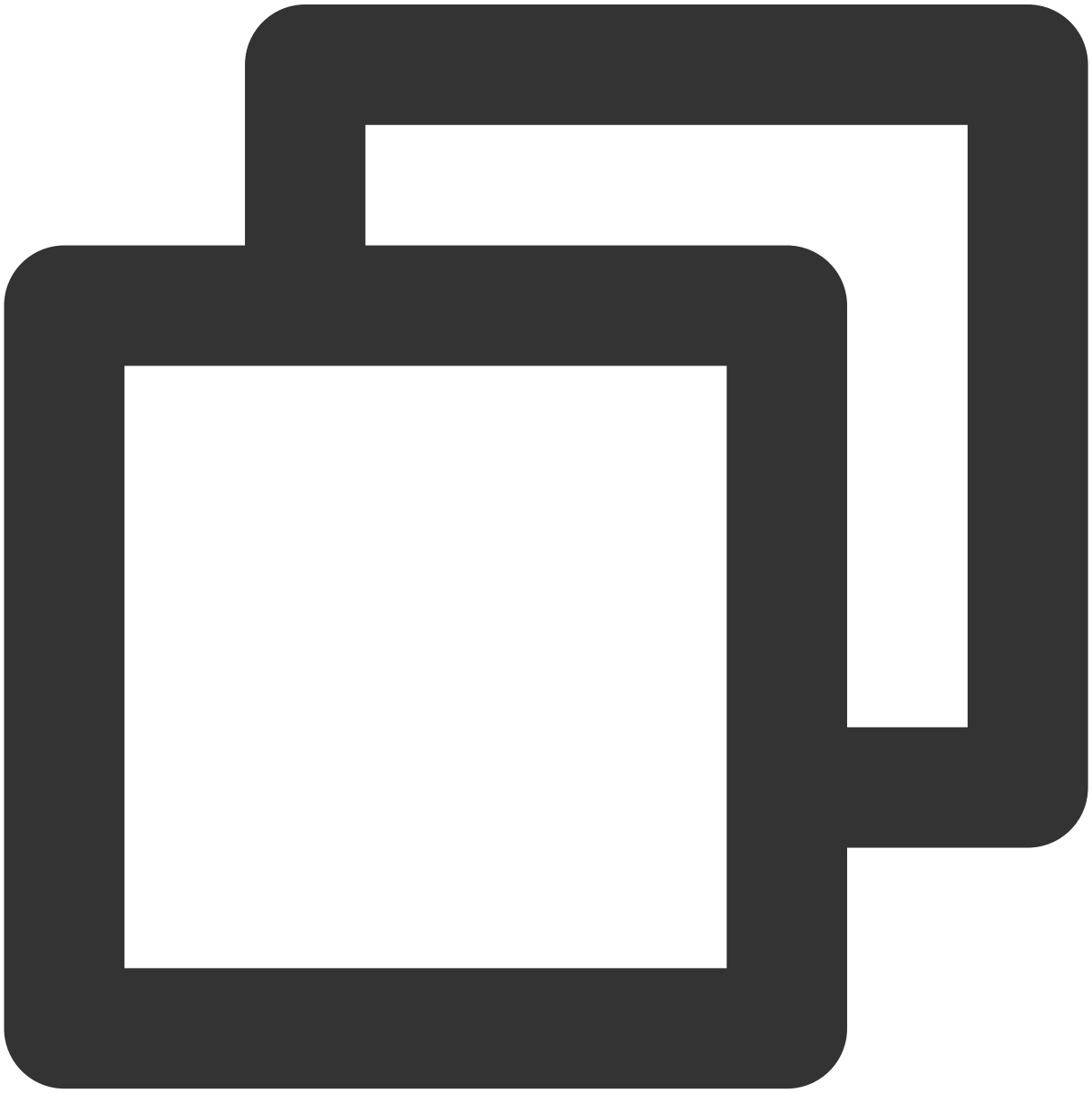
使用 SELECT 语句查询数据，期望自动改写并命中物化视图。可以通过查询结果里的统计数据，查看是否自动改写到了物化视图上。



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


查询结果
统计数据
运行历
数据扫描量
798B
自适应shuffle
shuflle字节数 --
溢出到COS --
缓存加速
ResultCache 缓存命中 否
FragmentCache命中率 --
Alluxio 缓存命中 --
物化视图加速
自动改写到
DataLakeCatalog.mv_test4.mv_stud
利用数据 446B
预处理：2.86s
执行：8.51s
获取结果：192.0

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

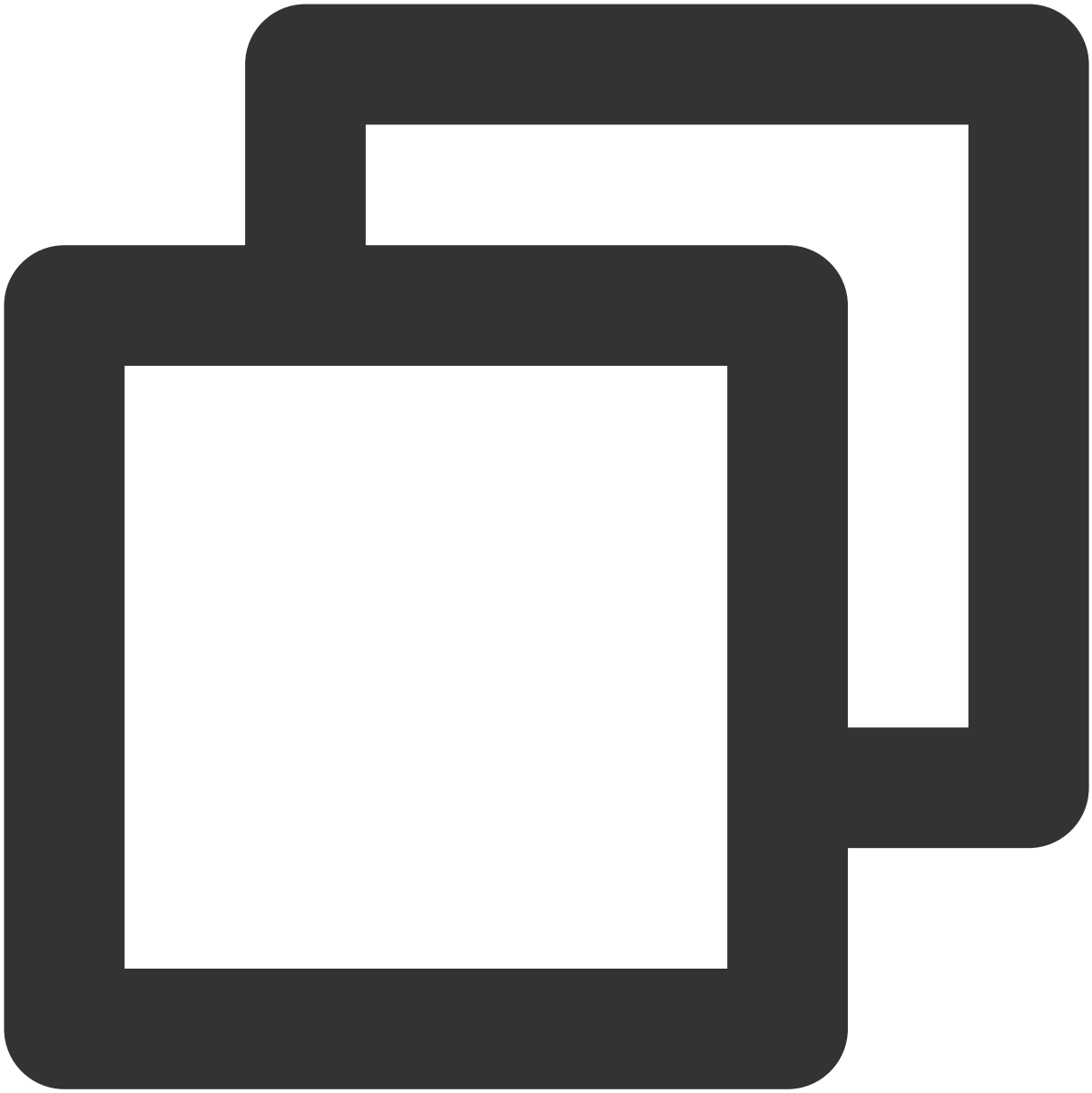
推荐您在以下场景使用映射物化视图：
当已经存在一个数据量较大的表，并且该表的查询性能较低时，可以通过映射物化视图来优化查询性能。
当需要保持物化视图的数据与现有表的数据保持一致，并且不需要自动刷新物化视图时，可以使用映射物化视图。

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

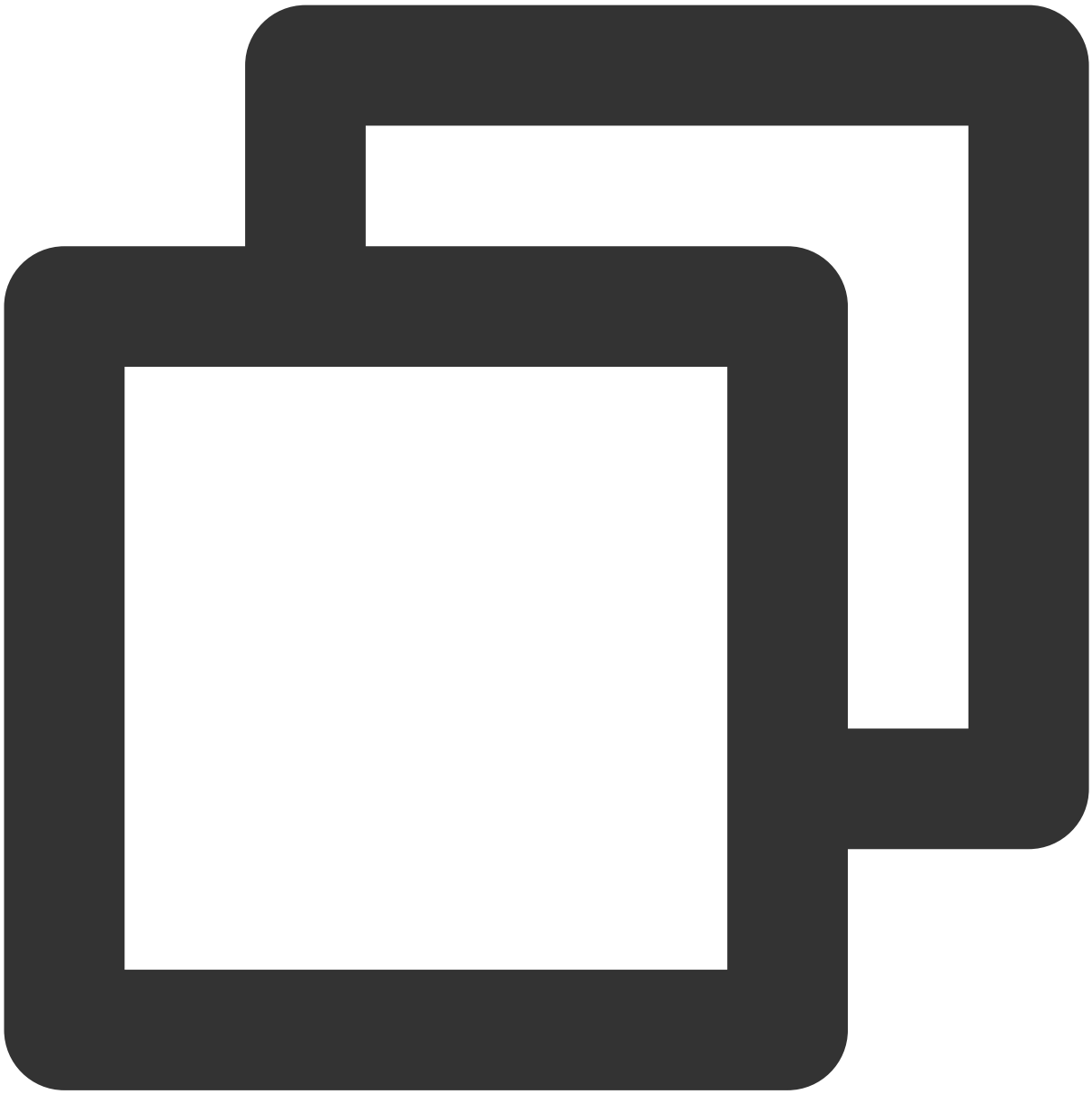
映射物化视图需要与待映射的表保持名称一致。以下例子先基于CTAS创建表，用于映射MV的创建。数据的准备可以参考普通物化视图中完整示例中的数据准备一节。



```
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

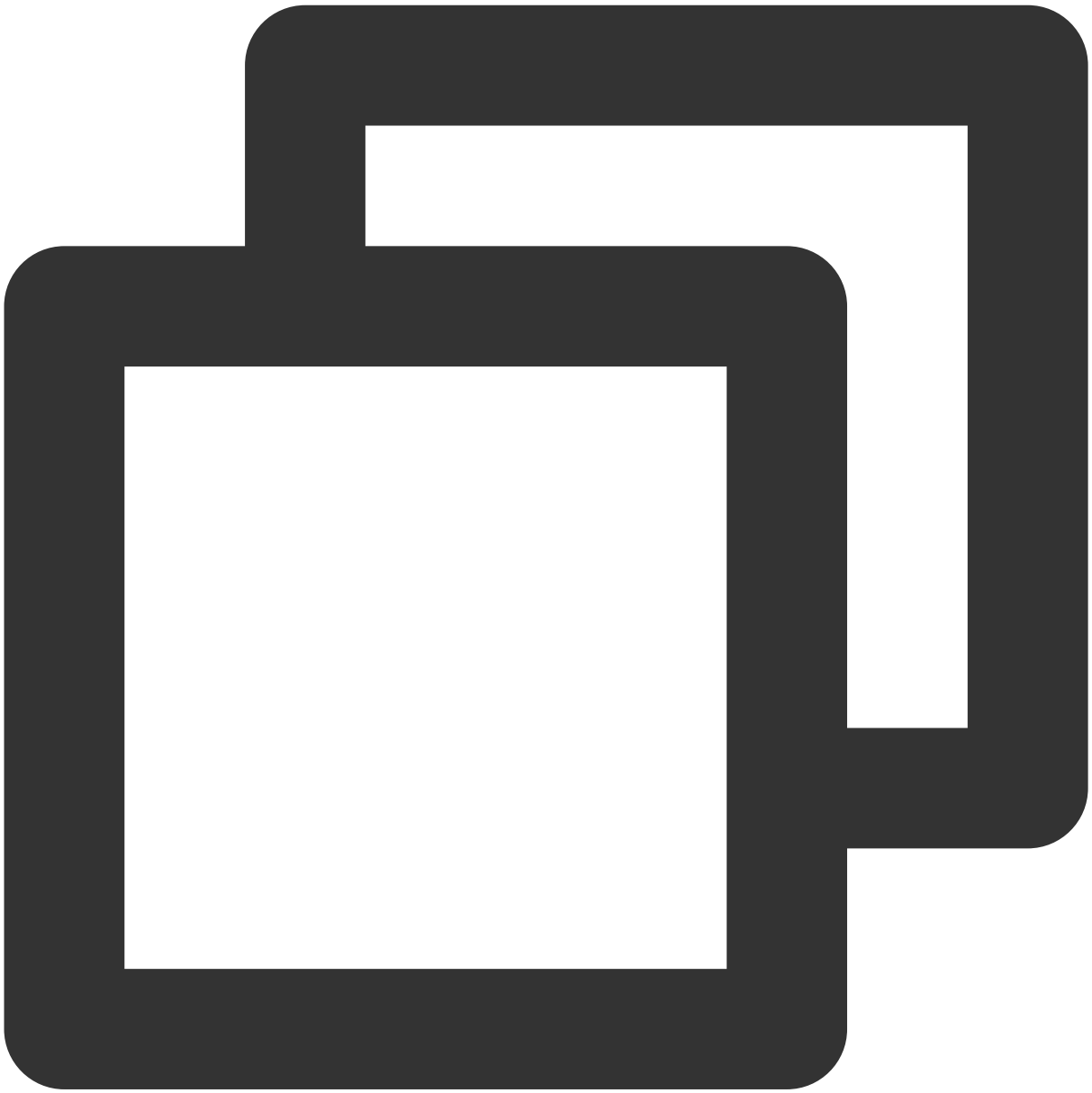
映射物化视图不支持 REFRESH 操作，即无法通过 REFRESH MATERIALIZED VIEW 语句来刷新物化视图的数据。因此，物化视图的数据只能与映射表的数据保持一致，无法自动更新。

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。

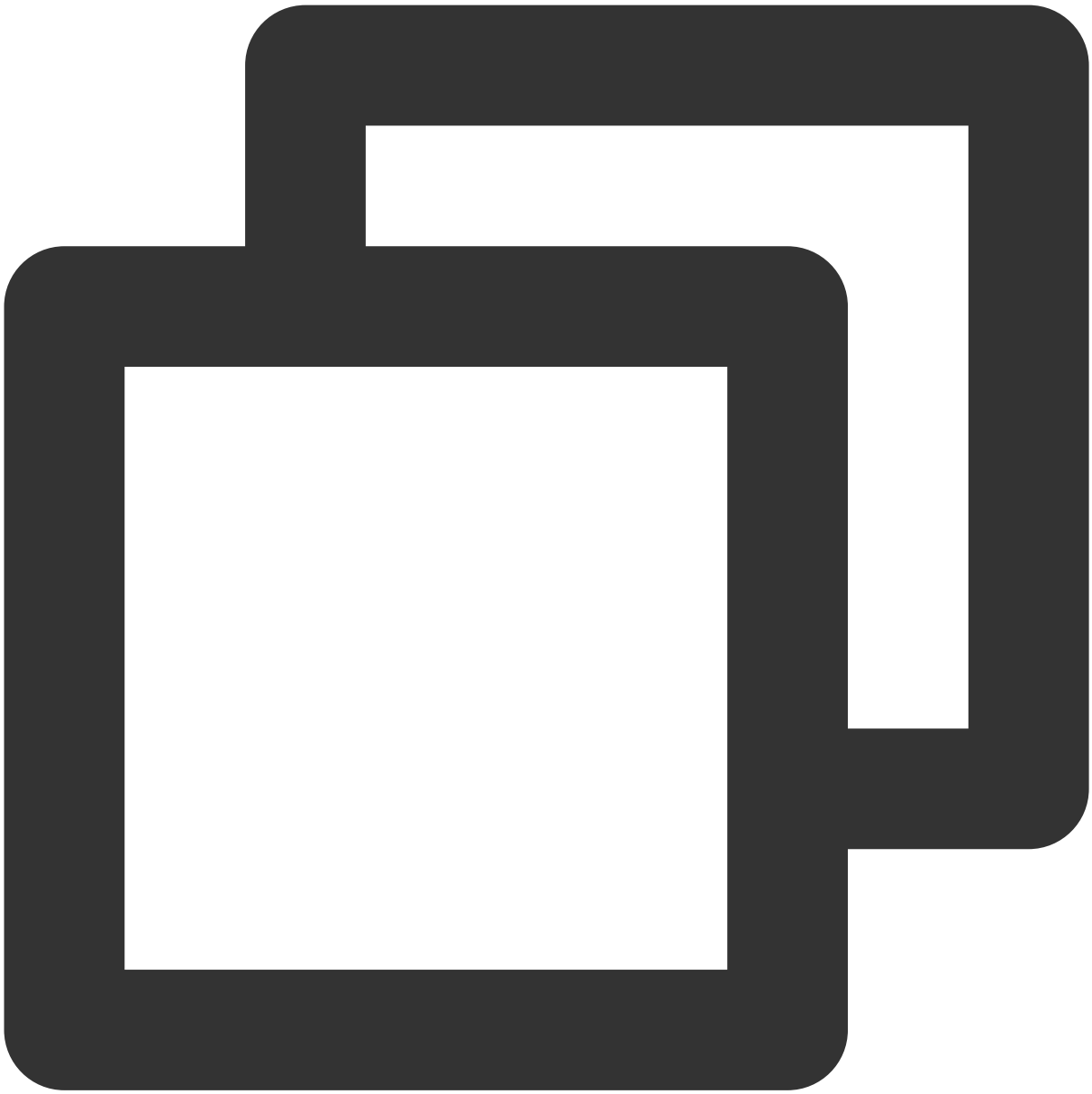
可以通过使用 Hint 或 TaskConf 参数来指定允许基于映射物化视图进行 SQL 改写。



```
--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。



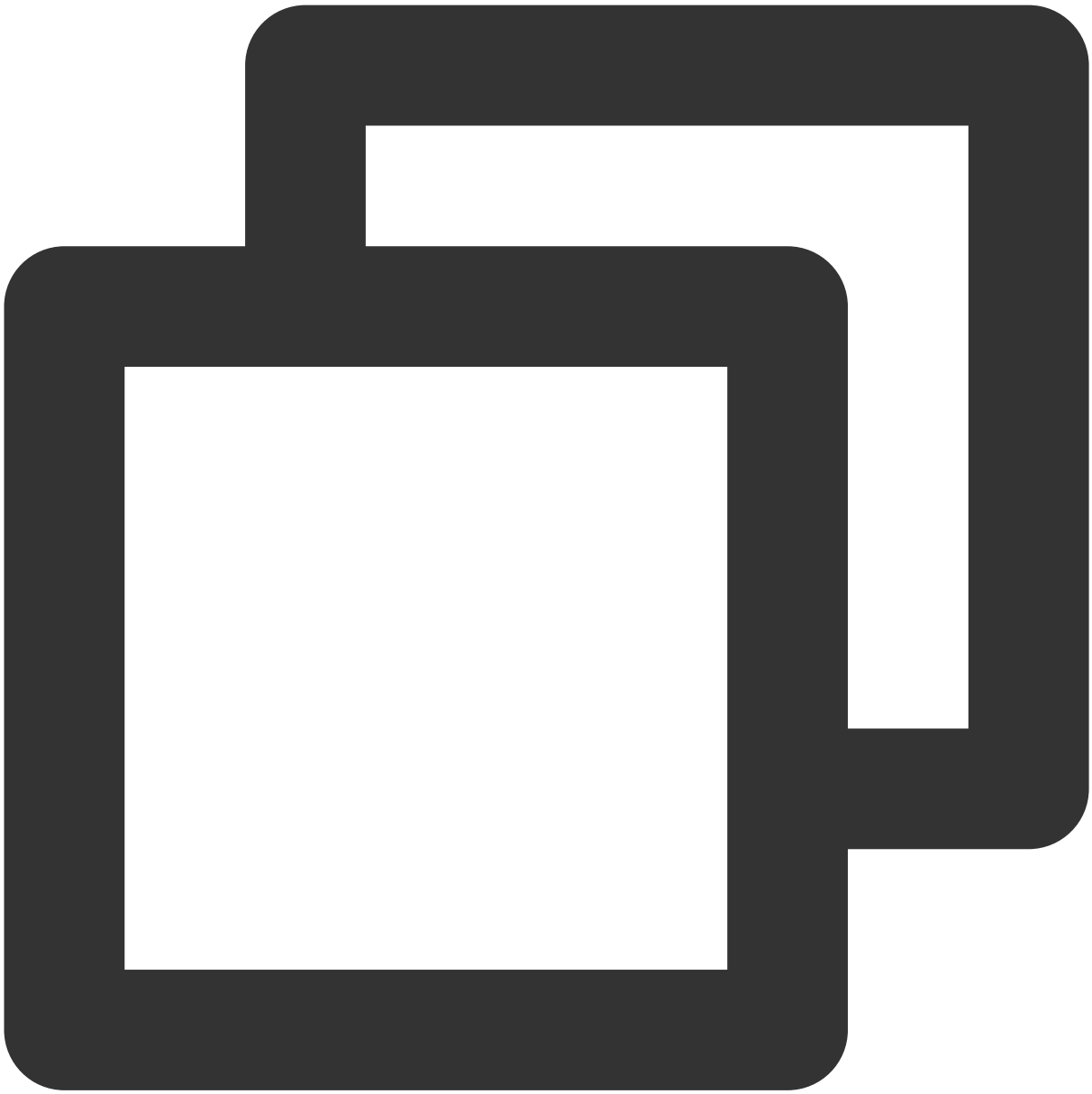
```
DROP MATERIALIZED VIEW link_mv_student;
DESCRIBE link_mv_student; --可查看源表还存在
```

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

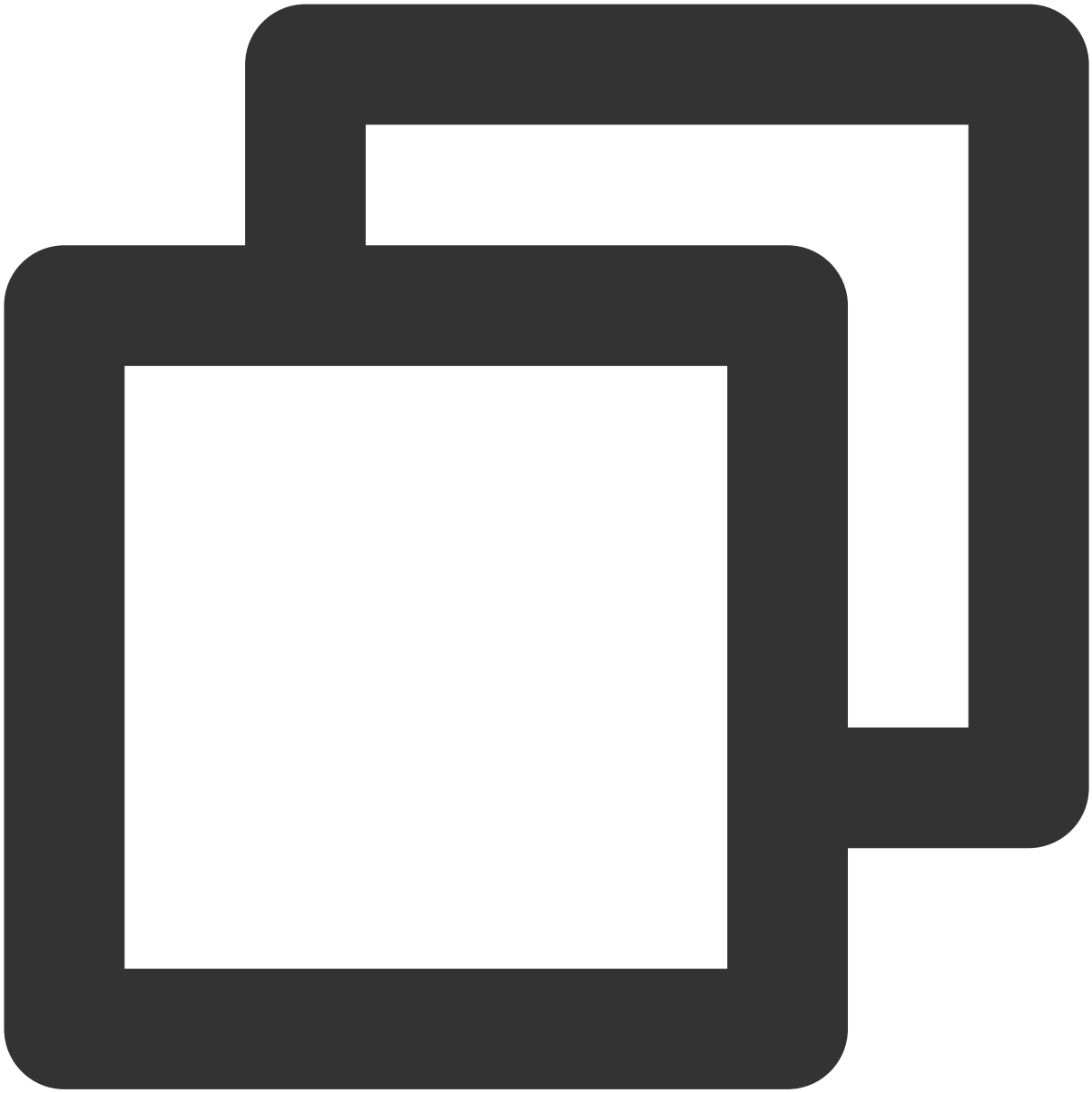
首先，需要准备初始化数据并创建Hive基表。使用 CREATE EXTERNAL TABLE 语句创建 Hive 基表，并通过 INSERT 语句手动插入数据。



```
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;
```

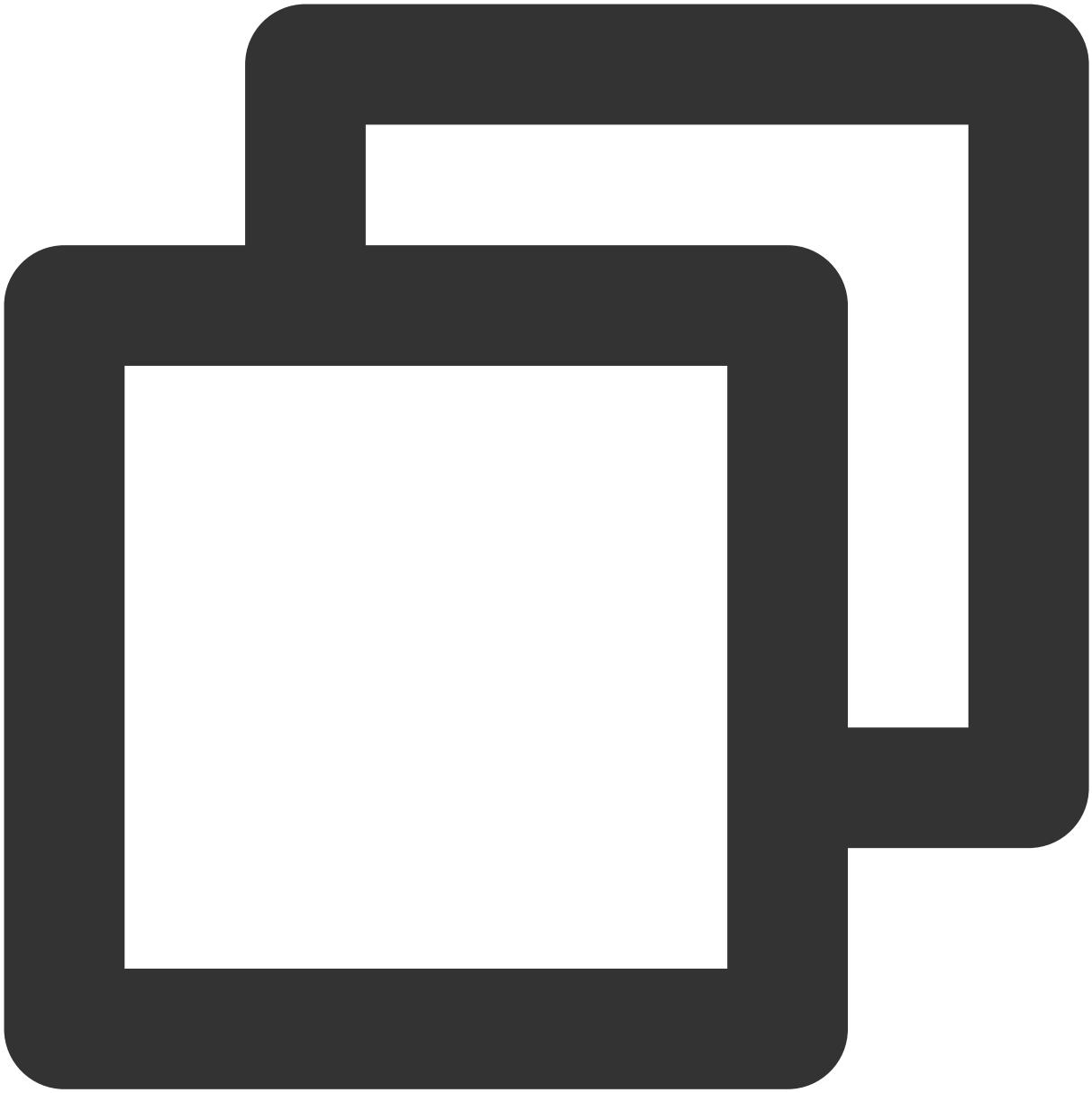
创建被映射的 Hive 外表

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';
```

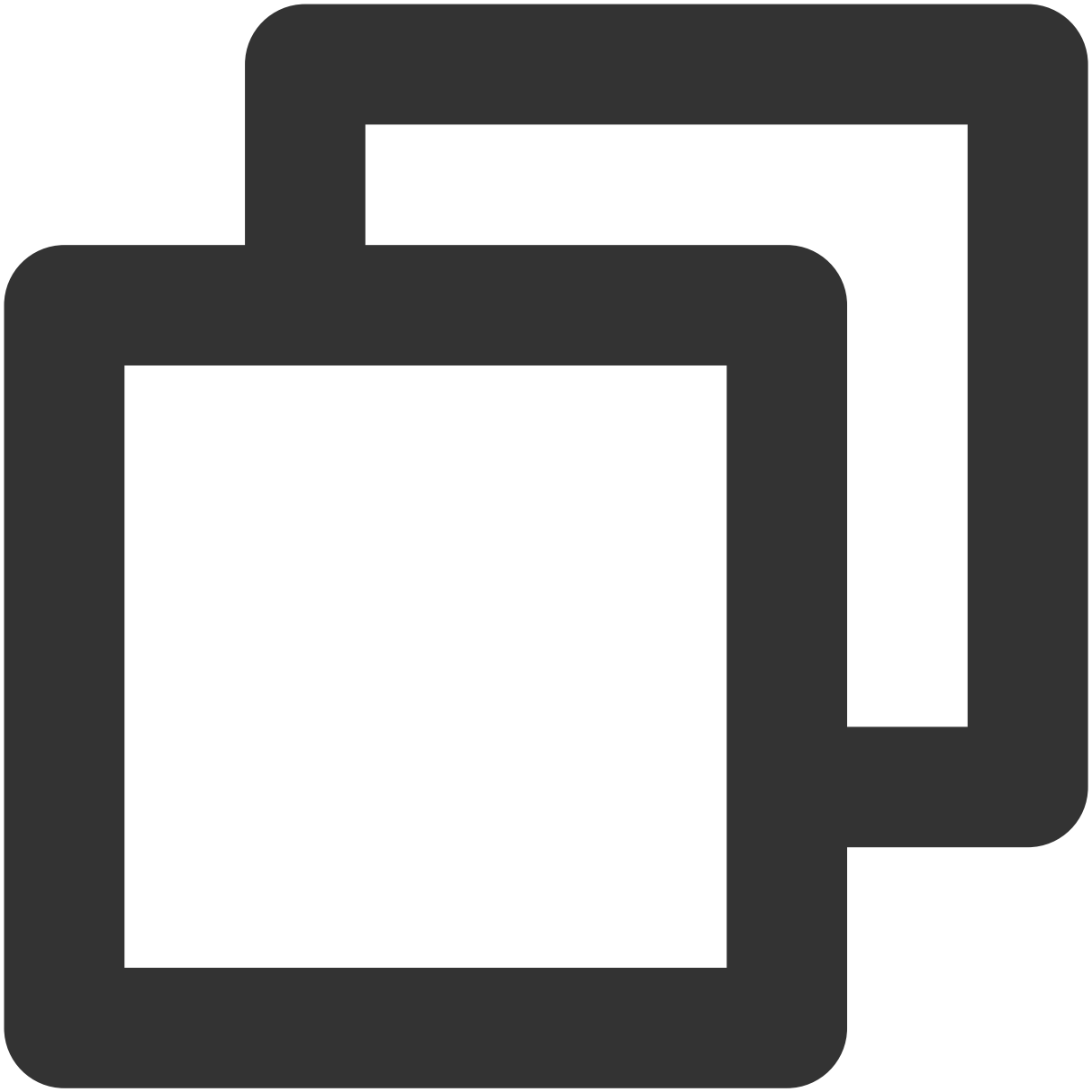
向映射表插入数据，使用 INSERT OVERWRITE 语句将查询结果插入到映射表中，确保映射表的数据与 Hive 基表的数据保持一致。



```
--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
);
```