数据湖计算

实践教程

产品文档

腾讯云

文档目录

实践教程
建表实践

最近更新时间：2025-03-12 18:06:08

DLC 数据湖计算支持创建原生表（Iceberg）、外表多种场景。具体可参考以下实践案例进行建表。

创建原生表（Iceberg）

Spark ETL 建表场景

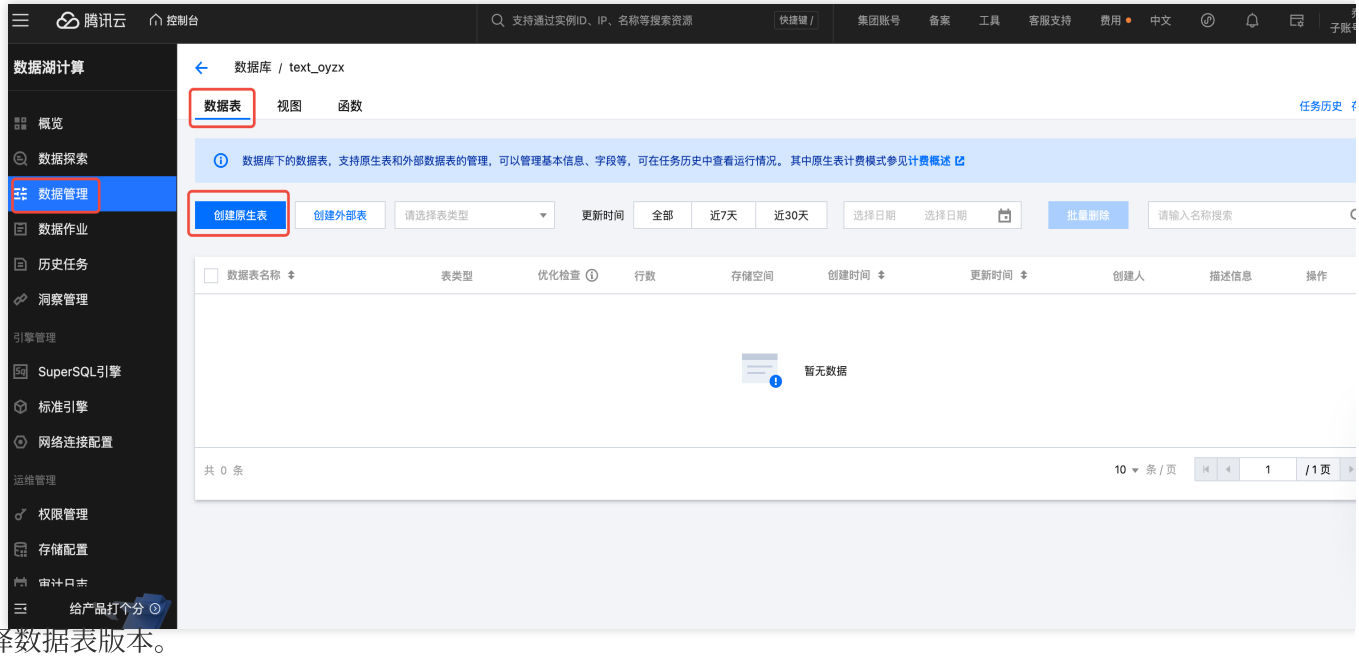
适用于：周期的进行 insert into、insert overwrite、merge into 等批作业操作。

```
/**
默认为 copy-on-write 模式，两种模式如果不确定，可不用配置，使用 copy-on-write 模式，merge-on
copy-on-write 适配场景： 查询性能相对更快，写入相对较慢，适用于周期的 ETL 任务或者较多数据量批量
merge-on-read 适配场景： 查询性能相对稍慢，写入更快，适用对写入性能有要求的场景，行级更新能力强，
*/
/** copy on write 表 */
CREATE TABLE dlc_db.iceberg_etl (
      id          INT,
      name        string,
      age         INT
  ) TBLPROPERTIES (
    'format-version' = '2',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.delete-after-commit.enabled' = 'true');

/** merge on read 表 */
CREATE TABLE dlc_db.iceberg_etl (
      id          INT,
      name        string,
      age         INT
  ) TBLPROPERTIES (
    'format-version' = '2',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.update.mode' = 'merge-on-read',
    'write.merge.mode' = 'merge-on-read',
    'write.delete.mode' = 'merge-on-read'
  );
```

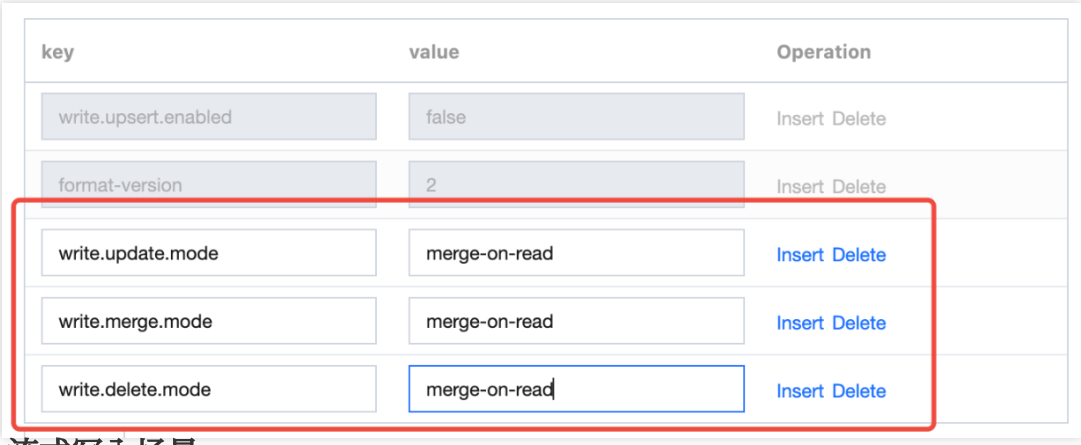
控制台创建：copy-on-write 模式

1. 单击创建原生表。



2. 选择数据表版本。

控制台创建：merge-on-read 模式（需要额外添加三个属性值）：



Flink 流式写入场景

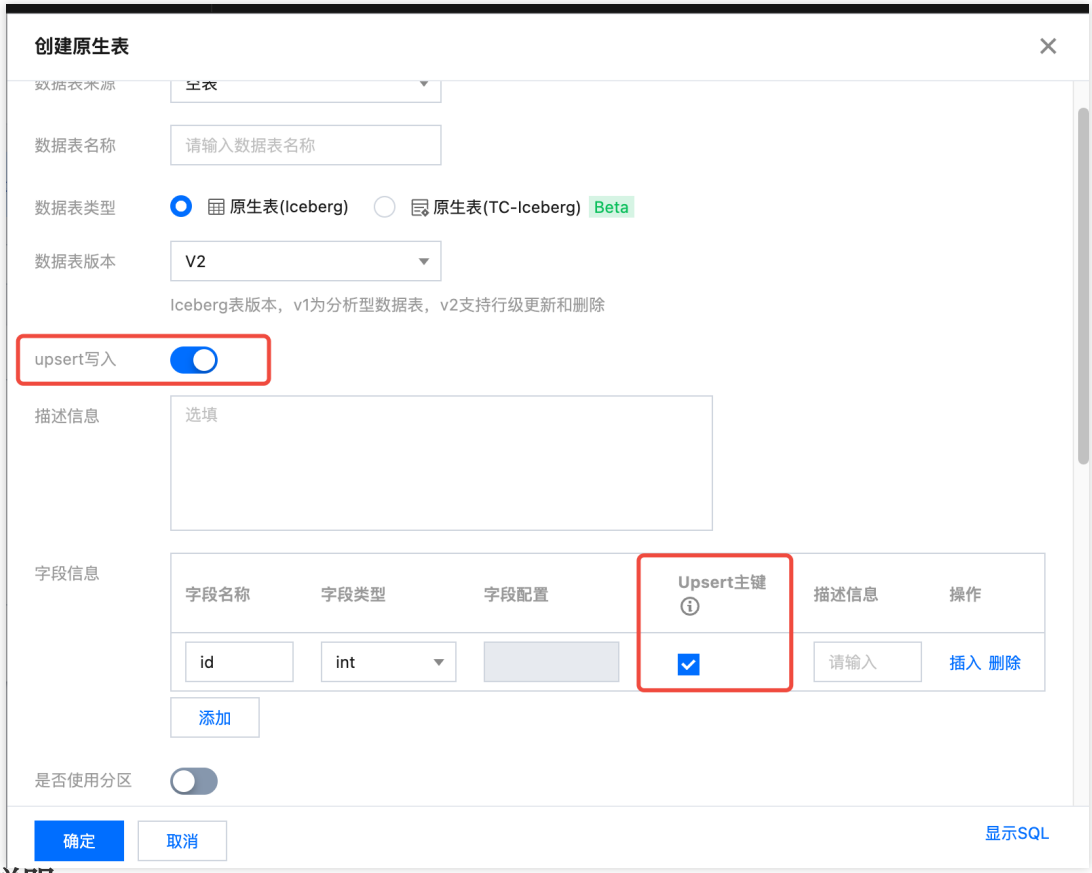
适用于：Oceanus (Flink 流式写入) 场景。

```
/** flink 流式写入主键为 id */
CREATE TABLE dlc_db.iceberg_cdc_by_id (
      id          INT,
      name        string,
      age         INT
  ) TBLPROPERTIES (
    'format-version' = '2',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.upsert.enabled' = 'true',
    'write.update.mode' = 'merge-on-read',
    'write.merge.mode' = 'merge-on-read',
    'write.delete.mode' = 'merge-on-read',
    'write.distribution-mode' = 'hash',
    'write.parquet.bloom-filter-enabled.column.id' = 'true',
    'dlc.ao.data.govern.sorted.keys' = 'id'
  );

/** flink 流式写入主键为 id,name 联合主键 */
CREATE TABLE dlc_db.iceberg_cdc_by_id_and_name (
      id          INT,
      name        string,
      age         INT
  ) TBLPROPERTIES (
    'format-version' = '2',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.upsert.enabled' = 'true',
    'write.update.mode' = 'merge-on-read',
    'write.merge.mode' = 'merge-on-read',
    'write.delete.mode' = 'merge-on-read',
    'write.distribution-mode' = 'hash',
```

```
    'write.parquet.bloom-filter-enabled.column.id' = 'true',
    'write.parquet.bloom-filter-enabled.column.name' = 'true',
    'dlc.ao.data.govern.sorted.keys' = 'id,name'
);
```

控制台创建：主键为 id 的表



配置说明

属性值	含义	配置指导
format-version	Iceberg 表版本，取值范围1、2。 标准引擎 Spark Standard-S 1.1、SuperSQL Spark 3.5 默认取值为2，其余场景默认值为1	建议配置为2
write.upsert.enabled	是否开启 upsert，取值为 true；不设置则为不开启	如果用户写入场景有 upsert，必须设置为 true
write.update.mode	更新模式	缺省为 copy-on-write 默认为 copy-on-write 模式，两种模式如果不确定，可不用配置，使用 copy-on-write 模式，

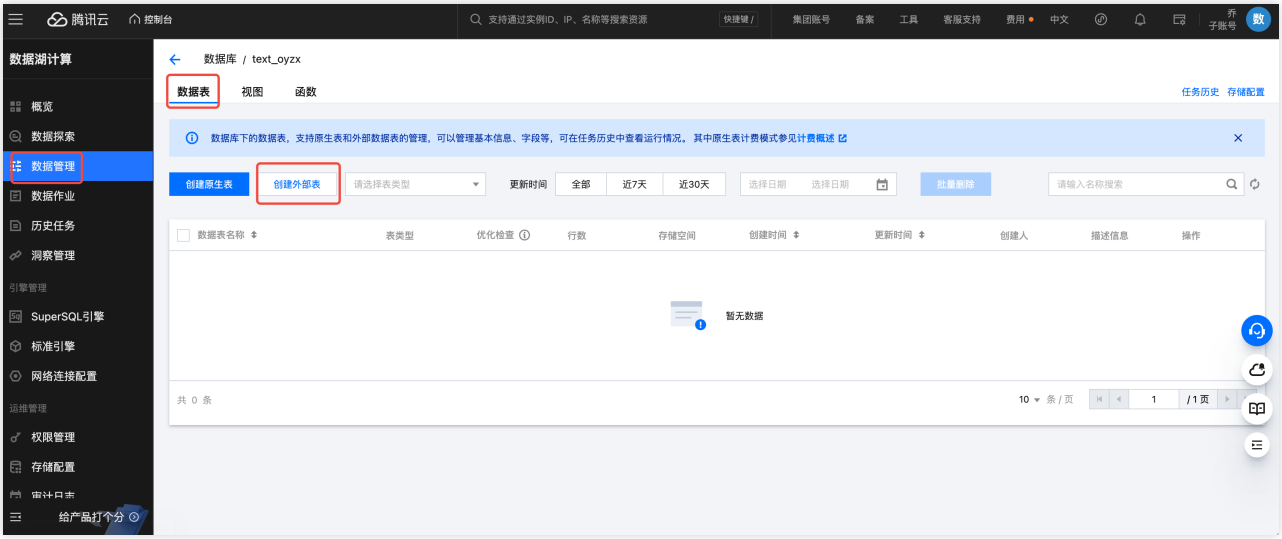
		merge-on-read 对行级更新场景有较大优化。 copy-on-write 适配场景：查询性能相对更快，写入相对较慢，适用于周期的 ETL 任务或者较多数据量批量更新操作场景。 merge-on-read 适配场景：查询性能相对稍慢，写入更快，适用对写入性能有要求的场景，行级更新能力强，对有频繁小范围（<10%） merge into/update/delete 或者 Oceanus(Flink 流式写入场景) 写入性能提升较大。
write.merge.mode	merge 模式	缺省为 copy-on-write 默认为 copy-on-write 模式，两种模式如果不确定，可不用配置，使用 copy-on-write 模式， merge-on-read 对行级更新场景有较大优化。 copy-on-write 适配场景：查询性能相对更快，写入相对较慢，适用于周期的 ETL 任务或者较多数据量批量更新操作场景。 merge-on-read 适配场景：查询性能相对稍慢，写入更快，适用对写入性能有要求的场景，行级更新能力强，对有频繁小范围（<10%） merge into/update/delete 或者 Oceanus(Flink 流式写入场景) 写入性能提升较大。
write.parquet.bloom-filter-enabled.column.{col}	仅用于Oceanus (Flink 流式写入) 场景，开启 bloom，取值为 true 表示开启，缺省不开启	Flink 流式写入场景必须开启，需要根据上游的主键进行配置；如上游有多个主键，最多取前两个；开启后可提升 MOR 查询和小文件合并性能
write.distribution-mode	写入模式	当取值为 hash 时，当数据写入时会自行进行 repartition，缺点是影响部分写入性能，建议保持默认行为； Flink 流式写入场景，建议配置为 hash ，可优化写入性能，其他场景建议不配置，保持默认值
write.metadata.delete-after-commit.enabled	开始 metadata 文件自动清理	强烈建议设置为 true，开启后 Iceberg 在产生快照时会自动清理历史的 metadata 文件，可避免大量的 metadata 文件堆积
write.metadata.previous-versions-max	设置默认保留的 metadata 文件数量	默认值为100，在某些特殊的情况下，用户可适当调整该值，需要配合 write.metadata.delete-after-commit.enabled 一起使用

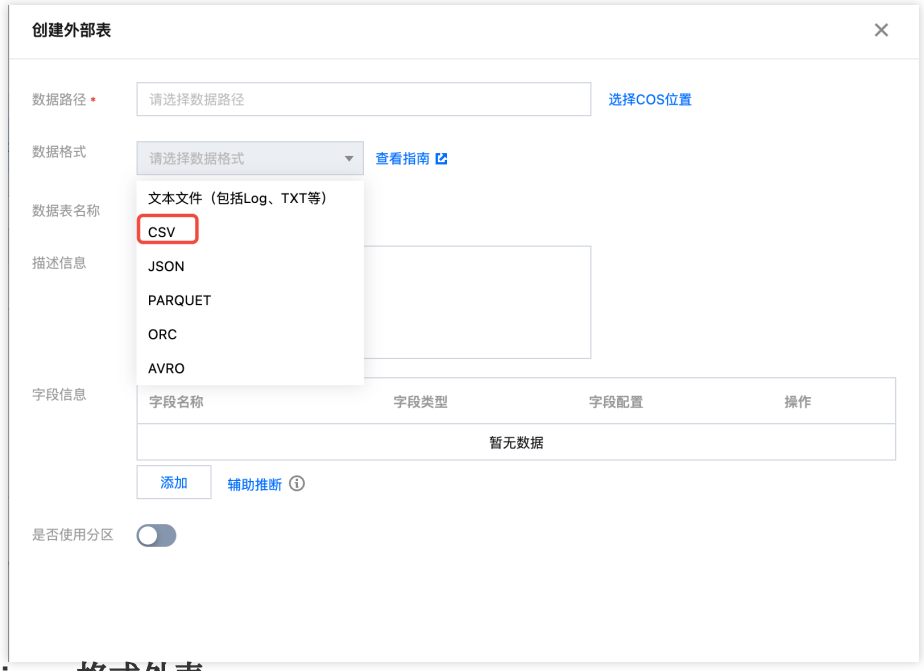
创建外表

创建 CSV 格式外表

```
/**
1. separatorChar: 分隔符，默认是 ，  。用于指定 CSV 文件中字段之间的分隔符，帮助 Hive 正确解析
1. quoteChar : 引用字符，默认是 " ，如果原始文件无引用字符，可以用默认值。引用字符作用是 可以帮
1. LOCATION:  需要修改为对应 cos 上的存储路径
1. TBLPROPERTIES 中的 skip.header.line.count 表属性： 默认0，配置1 代表跳过一行，该属性用于
*/
CREATE EXTERNAL TABLE IF NOT EXISTS  dlc_db.`csv_tb`(
    `id` int,
    `name` string)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  'quoteChar'='"',
  'separatorChar'=',')
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
  'cosn://your_cos_location'
TBLPROPERTIES (
  'skip.header.line.count'='1');
```

控制台创建





创建 json 格式外表

```
/**
 LOCATION： 指向对应的 cos 存储目录，其他保持原样
*/
CREATE EXTERNAL TABLE IF NOT EXISTS dlc_db.json_demo
(`id` bigint, `name` string)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE
LOCATION 'cosn://your_cos_location'
```

json 文件内容示例，每行是一个独立的 json 串：

```
{"id":1,"name":"tom"}
{"id":2,"name":"tony"}
```

创建 parquet 格式外表

```
/**
LOCATION： 指向对应的 cos 存储目录,其他保持原样
*/
CREATE EXTERNAL TABLE IF NOT EXISTS dlc_db.parquet_demo
 (`id` int, `name` string)
 PARTITIONED BY (`dt` string)
 STORED AS PARQUET LOCATION 'cosn://your_cos_location';
```

创建 ORC 格式外表

```
/**
LOCATION： 指向对应的 cos 存储目录,其他保持原样
*/
CREATE EXTERNAL TABLE IF NOT EXISTS dlc_db.orc_demo
(`id` int,`name` string)
 PARTITIONED BY (`dt` string)
 STORED AS ORC LOCATION 'cosn://your_cos_location'
```

创建 AVRO 格式外表

```
/**
LOCATION： 指向对应的 cos 存储目录,其他保持原样
*/
CREATE EXTERNAL TABLE IF NOT EXISTS dlc_db.avro_demo
(`id` int,`name` string)
 PARTITIONED BY (`dt` string)
 STORED AS ORC LOCATION 'cosn://your_cos_location'
```

补充

列类型和分区列

参考 CREATE TABLE 参数章节。

注意：

二进制类型字段 binary 在执行 select 查询语句时可能会遇到如下报错。原因是引擎默认将结果集写入 csv 文件，暂不支持将二进制数据写入 csv 文件。

```
Error operating ExecuteStatement: org.apache.spark.sql.AnalysisException:
[UNSUPPORTED_DATA_TYPE_FOR_DATASOURCE] The CSV datasource doesn't support the column `bno` of the type
"BINARY". at
```

解决方法（支持任务级配置）：

1. 更改保存结果的文件格式：kyuubi.operation.result.saveToFile.format=parquet（设置存储文件格式，可选项：parquet, orc）。
2. 更改配置不保存结果到 cos：kyuubi.operation.result.saveToFile.enabled=false。

复杂列类型

```
/**
LOCATION： 指向对应的 cos 存储目录
其他保持原样
```

```
*/
CREATE EXTERNAL TABLE dlc_db.orc_demo_with_complex_type(
    col_bigint bigint COMMENT 'id number',
    col_int int,
    col_struct struct<x: double, y: double>,
    col_array array<struct<x: double, y: double>>,
    col_map map<struct<x: int>, struct<a: int>>,
    col_decimal DECIMAL(10,2),
    col_float FLOAT,
    col_double DOUBLE,
    col_string STRING,
    col_boolean BOOLEAN,
    col_date DATE,
    col_timestamp TIMESTAMP
)
PARTITIONED BY (`dt` string)
STORED AS ORC LOCATION 'cosn://your_cos_location';
```

注意：

1. avro 格式数据源不支持 map 类型和 array 类型嵌套 struct 结构。
2. avro 格式数据源 map 类型字段的 key 仅支持 string 类型。
3. csv 格式数据源使用 struct、array、map 类型字段时可能会遇到如下报错，原因是引擎对数据格式做了强校验。
解决方法：解除引擎的强校验设置，设置引擎静态参数 spark.sql.storeAssignmentPolicy=legacy。

复杂分区类型

1. LOCATION: 需要修改为对应 cos 上的存储路径
2. 支持的分区字段类型有TINYINT，SMALLINT，INT，BIGINT，DECIMAL，FLOAT（不推荐，建议使用DECIMAL），DOUBLE（不推荐，建议使用 DECIMAL），STRING，BOOLEAN，DATE，TIMESTAMP。

```
CREATE EXTERNAL TABLE dlc_db.orc_demo_with_complex_partition(
    col_int int
)
PARTITIONED BY (
    pt_tinyint TINYINT,
    pt_smallint SMALLINT,
    pt_decimal DECIMAL(10,2),
    pt_string STRING,
    pt_date DATE,
    pt_timestamp TIMESTAMP )
STORED AS ORC LOCATION 'cosn://lcl-bucket-1305424723/dlc/orc_demo_with_complex_part
```

注意：

hive 类型表分区名之和不能超过767个字符。

元数据不区分大小写

元数据中的表名和列名在使用时不区分大小写，但在数据管理界面展示时会保留创建时的原始大小写格式。

使用 Apache Airflow 调度 DLC 引擎提交任务

最近更新时间：2025-05-22 15:45:23

本文介绍 DLC 对 Apache Airflow 调度工具的支持，并提供示例来演示如何使用 Apache Airflow 运行 DLC 不同种类的引擎任务。

背景信息

Apache Airflow 是一款由 Airbnb 开源的调度工具，用 Python 编写，采用有向无环图（DAG）的方式来定义和调度一组有依赖关系的作业。它支持 Python 编写的子作业，并提供多种操作器（Operators）来执行任务，如 Bash 命令、Python 函数、SQL 查询和 Spark 作业等，具备很高的灵活性和可扩展性。Apache Airflow 广泛应用于数据工程、数据处理和工作流自动化等领域。借助 Apache Airflow 提供的丰富功能和可视化界面，用户可以轻松监控和管理工作流的状态和执行情况。更多关于 Apache Airflow 信息，请参见 Apache Airflow。

前提条件

1. Apache Airflow 环境准备。
2. 安装并启动 Apache Airflow，更多安装及启动 Apache Airflow 操作，请参见Apache Airflow 快速入门。
3. 安装 jaydebeapi 依赖包， pip install jaydebeapi。
4. 数据湖计算 DLC 环境准备。
5. 开通数据湖计算 DLC 引擎服务。
6. 如使用标准 Spark 引擎，准备好 Hive JDBC 驱动，点击下载 hive-jdbc-3.1.2-standalone.jar。
7. 如使用标准 Presto 引擎，准备好 Presto JDBC 驱动，点击下载 presto-jdbc-0.284.jar。
8. 如使用 SuperSQL 引擎，准备好 DLC JDBC 驱动，点击下载 JDBC 驱动。

关键步骤

创建 Connection 和调度任务

在 Apache Airflow 工作目录下新建 dags 目录，在 dags 目录下新建调度脚本并保存为.py 文件，例如本文建立调度脚本/root/airflow/dags/airflow-dlc-test.py 如下所示：

```
import time
from datetime import datetime, timedelta

import jaydebeapi
from airflow import DAG
```

```
from airflow.operators.python_operator import PythonOperator

jdbc_url='jdbc:dlc:dlc.tencentcloudapi.com?task_type=SparkSQLTask&database_name={da
user = 'xxx'
pwd = 'xxx'
dirver = 'com.tencent.cloud.dlc.jdbc.DlcDriver'
jar_file = '/root/airflow/jars/dlc-jdbc-2.5.3-jar-with-dependencies.jar'

def createTable():
    sqlStr = 'create table if not exists db.tb1 (c1 int, c2 string)'
    conn = jaydebeapi.connect(dirver, jdbc_url, [user, pwd], jar_file)
    curs = conn.cursor()
    curs.execute(sqlStr)
    rows = curs.rowcount.real
    if rows != 0:
        result = curs.fetchall()
        print(result)
    curs.close()
    conn.close()


def insertValues():
    sqlStr = "insert into db.tb1 values (111, 'this is test')"
    conn = jaydebeapi.connect(dirver,jdbc_url, [user, pwd], jar_file)
    curs = conn.cursor()
    curs.execute(sqlStr)
    rows = curs.rowcount.real
    if rows != 0:
        result = curs.fetchall()
        print(result)
    curs.close()
    conn.close()


def selectColums():
    sqlStr = 'select * from db.tb1'
    conn = jaydebeapi.connect(dirver, jdbc_url, [user, pwd], jar_file)
    curs = conn.cursor()
    curs.execute(sqlStr)
    rows = curs.rowcount.real
    if rows != 0:
        result = curs.fetchall()
        print(result)
    curs.close()
    conn.close()
```

```
def get_time():
    print('当前时间是:', datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
    return time.time()


default_args = {
    'owner': 'tencent',  # 拥有者名称
    'start_date': datetime(2024, 11, 1),  # 第一次开始执行的时间，为 UTC 时间
    'retries': 2,  # 失败重试次数
    'retry_delay': timedelta(minutes=1),  # 失败重试间隔
}
dag = DAG(
    dag_id='airflow_dlc_test',  # DAG id ,必须完全由字母、数字、下划线组成
    default_args=default_args,  # 外部定义的 dic 格式的参数
    schedule_interval=timedelta(minutes=1),  # 定义DAG运行的频率,可以配置天、周、小时、分
    catchup=False  # 执行DAG时，将开始时间到目前所有该执行的任务都执行，默认为True
)

t1 = PythonOperator(
    task_id='create_table',
    python_callable=createTable,
    dag=dag)

t2 = PythonOperator(
    task_id='insert_values',
    python_callable=insertValues,
    dag=dag)

t3 = PythonOperator(
    task_id='select_values',
    python_callable=selectColums,
    dag=dag)

t4 = PythonOperator(
    task_id='print_time',
    python_callable=get_time,
    dag=dag)

t1 >> t2 >> [t3, t4]
```

参数说明：

参数	说明
jdbc_url	jdbc 的连接地址以及配置参数。详情请参见 Hive JDBC 访问、Presto JDBC 访问和 DLC

	JDBC 访问
user	SecretId
pwd	SecretKey
dirver	加载 JDBC 驱动。详情请参见 Hive JDBC 访问、Presto JDBC 访问和 DLC JDBC 访问
jar_file	驱动 jar 包的存放路径，需替换对应引擎 JDBC 驱动 jar 包存放绝对路径。详情请参见 Hive JDBC 访问、Presto JDBC 访问和 DLC JDBC 访问

运行调度任务

您可以进入 Web 界面，在 DAGs 页签，查找到提交的调度流程并启动调度。

查看任务运行结果

DLC + Wedata 实现机器学习实践教程

最近更新时间：2025-06-25 16:03:40

数据湖计算 DLC 支持通过 spark 引擎创建机器学习资源组的方式，协助用户进行模型训练等机器学习场景。
通过本篇文档，您可根据我们提供的 demo 数据集与代码示例，体验在 Scikit-learn 的框架下进行模型训练的实践体验。

说明：

资源组：Spark 标准引擎计算资源的二级队列划分，资源组隶属于父级标准引擎且同一引擎下的资源组彼此资源共享。
DLC Spark 标准引擎的计算单元（CU）可按需被划分到多个资源组中，并设置每个资源组可使用CU数量的最小值和上限、启停策略、并发数和动静态参数等，从而满足多租户、多任务等复杂场景下的计算资源隔离与工作负载的高效管理。实现不同类别任务间的资源隔离，避免个别大查询将资源长期抢占。
目前 DLC 机器学习资源组、WeData Notebook 探索、机器学习均为**开白功能**，如需使用，请 提交工单 联系 DLC 与 WeData 团队开通机器学习资源组、Notebook、MLFlow 服务。

开通账号与产品

开通账号与产品

DLC 账号与产品开通功能均需腾讯云主账号进行开通，主账号完成后，默认主账号下所有子账号均可使用。如果需要调整，可通过 CAM 功能进行调整。具体操作指引请参见 新用户开通全流程。
Wedata 账号与产品开通请参见 准备工作、数据湖计算 DLC（DLC）。
功能和 MFlow 服务开通均为主账号粒度，一个主账号操作完成后，该主账号下的所有子账号均可使用。
需要提供客户的地域信息、APPID、主账号UIN、VPC ID和子网ID，其中 VPC 和子网信息用于 MFlow 服务的网络打通操作。

说明：

因产品中多个功能需要进行网络打通操作，为确保网络连通性，建议后续购买执行资源组、创建 Notebook 工作空间等操作都在这一个 VPC 和子网中进行。

配置数据访问策略

数据访问策略（CAM role arn）是为了保障数据作业运行过程中访问的数据源及对象存储 COS 上的数据安全，用户在访问管理（CAM）上对数据访问权限进行配置的策略。
在数据湖计算 DLC 中配置数据作业时，需指定对应的数据访问策略，以保证数据安全。
配置方式请参见 配置数据访问策略。

在 DLC 购买计算资源

在产品服务开通完成后，您可先通过数据湖计算 DLC 购买计算资源。如果您需要使用机器学习功能，**请确认购买的引擎类型为：标准引擎-spark，内核版本为：Standard-S 1.1。**

1. 进入数据湖计算DLC > 标准引擎页面。
2. 选择“创建资源”。
3. 购买标准引擎-spark，内核版本选择：Standard1.1。

说明：

1. 购买账号需要有财务权限或是主账号进行购买。
2. 计费模式您可根据您的业务场景进行选择。
3. 集群规格建议选择64CU以上。
4. 购买成功后，首次启动会有数分钟的等待时间，如长时间未能完成启动，请 提交工单。

创建机器学习资源组

标准引擎购买完成后，返回引擎管理页面，您需要在该引擎下，创建机器学习资源组，才能开始进行机器学习相关功能。

注意：

资源组创建完成后，不支持编辑更改，可通过删除重新创建的方式进行管理。

1. 单击**管理资源组**。

2. 进入资源组页面后，单击左上角**创建资源组按钮。**
3. 创建机器学习资源组类型。

说明：

数据湖计算 DLC AI 资源组目前支持通过：Scikit-learn（1.6.0）、TensorFlow（2.18.0）、PyTorch（2.5.1）、Python（3）、Spark MLlib（3.5）框架执行机器学习。

业务场景选择：机器学习。

框架类型：您可根据实际业务场景，选择适合的框架创建。

如果您需要体验我们提供的demo，请选择ML开源框架，镜像包请选择：scikit-learn-v1.6.0。

资源配置：可按需选择。

配置完成后，单击确认返回资源组列表页。数分钟后，可单击列表页上方刷新按钮进行确认。

上传机器学习数据集至 COS

如果您需要通过数据湖计算 DLC 与 Wedata 的方式进行机器学习，目前仅支持数据上云的方式进行交互。我们推荐您使用对象存储进行组合使用。

说明：

目前暂支持通过 spark 直接读取 cos 数据。

如果有其他框架诉求，目前绕行方案：先把数据上传至COS > 下载至本地生成本地文件，再进行学习操作。该绕行方案，可能会出现上传及下载时间较长的情况出现。

该功能优化我们正在开发支持中。

1. 开通 对象存储 产品服务并创建存储桶，开通方式请参见 控制台快速入门。
2. 登录 对象存储，选择存储桶，上传该数据集。
3. 上传完成后，进入元数据管理，单击创建数据目录，或使用已有数据目录上传外表。
4. 进入 DLC 控制台 > 元数据管理，单击数据库 tab 页。
5. 单击创建数据库，命名为：database_testnotebook。
6. 进入创建好的数据库，单击创建外表。

注意：

请留意您上传外表的数据库表名称，在 Notebook 上会通过 select 调用库、表名称。

7. 选择 cos 存储桶路径，找到 demo 数据集。
8. 数据格式选择为 csv，并进行相关配置。

9. 创建表名称为：demo_test_sklearn。
10. 创建完成后，单击确认返回。

您还可以通过 SQL 执行，详情请参见 建表实践 > 创建 CSV 格式外表。

```
CREATE TABLE database_testnotebook.demo_test_sklearn (
  at STRING COMMENT 'from deserializer',
  v STRING COMMENT 'from deserializer',
  ap STRING COMMENT 'from deserializer',
  rh STRING COMMENT 'from deserializer',
  pe STRING COMMENT 'from deserializer')
USING csv
LOCATION 'cosn://your cos location'
```

前往 Wedata-Notebook 功能进行 demo 实践

资源组与 demo 数据集创建完成后，前往 Wedata 通过 Notebook 和 MLFlow 进行模型训练实践。

创建 WeData 项目并关联 DLC 引擎

1. 创建项目或选择已有的项目，详情请参见 项目列表。

2. 在配置存算引擎中选择所需的 DLC 引擎。

购买执行资源组并关联项目

如果您需要在编排空间中周期性调度 Notebook 任务，请购买调度资源组并与指定项目进行关联。详情请参见 调度资源组配置。

操作步骤：

1. 进入“执行资源组 > 调度资源组 > 标准调度资源组”，单击**创建**。
2. 资源组配置。

地域：调度资源组所在地域需要与存算引擎所在地域保持一致，例如购买了国际站-新加坡地域的DLC引擎，则需要购买相同地域的调度资源组。

VPC 和子网：建议直接选择1.1中的 VPC 和子网，如选择其他 VPC 和子网，需要确保所选 VPC 和子网与1.1中的 VPC 和子网之间网络互通。

规格：按照任务量进行选择。

3. 创建完成后，在资源组列表的操作栏单击“关联项目”，将该调度资源组与所需使用的项目进行关联。

创建 Notebook 工作空间

1. 在项目中，选择数据治理功能，单击 Notebook 功能，并创建或使用已有工作空间。

2. 创建工作空间时，请选择购买**标准 spark 引擎，standard1.1版本的引擎，勾选机器学习选项以及 MFlow 服务。**

基本信息	属性项名称	属性项配置
	引擎	选择一个您需要使用 Notebook 任务访问的 DLC 引擎。 当前项目项目管理中中绑定的 DLC 引擎。
	DLC 数据引擎	选择一个您需要使用 Notebook 任务访问的 DLC 数据引擎。
	机器学习	如果您选择的 DLC 数据引擎中含有“机器学习”类型的资源组，则会出现该选项，并默认选中。
	网络	建议直接选择1.1中的 VPC 和子网，如选择其他 VPC 和子网，需要确保所选 VPC 和子网与1.1中的 VPC 和子网之间网络互通。
	RoleArn	RoleArn为DLC引擎访问对象存储COS的数据访问策略(CAM role arn)，详情请参见 配置数据访问策略。
高级配置	MFlow 服务	使用 MFlow 管理实验和模型，默认为不勾选。 勾选后，则在 Notebook 任务中使用MFlow函数创建实验和机器学习，均会上报到1.1中部署的 MFlow 服务中，后续可以在机器学习 > 实验管理、模型管理中进行查看。

创建 Notebook 文件

在左侧资源管理器可以创建文件夹和Notebook文件，注意：Notebook文件需要以（.ipynb）结尾。在资源管理器中，预先内置了三个大数据系列教程，支持用户开箱即用。

选择内核（kernel）

1. 单击**选择内核**。

2. 在弹出的下拉选项中选择“DLC 资源组”。

3. 在下一级选项中选择 DLC 数据引擎中的您创建的 Scikit-learn 资源组。

例如，上图选择的名为“machine learning -（测试 wedata_sklearn 资源组）”的资源组，与 DLC 数据引擎中的资源组名称一致：

运行 Notebook 文件

1. 确认初始化配置。

2. 执行实践教程：利用公开数据鸢尾花集进行演示，采用逻辑回归模型对不同类型的花分类，并对分类结果可视化。

注意：

在运行模型前，需安装必要的tencentcloud-dlc-connector，并完成相应配置。

```
#安装驱动
!pip install tencentcloud-dlc-connector
!pip  install --upgrade 'sqlalchemy<2.0'

#安装版本
!pip install --upgrade pandas==2.2.3
!pip install numpy
!pip install matplotlib
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import tdlc_connector
from tdlc_connector import constants
import mlflow
mlflow.sklearn.autolog()

#使用 tdlc-connector 按照表方式访问
```

```
conn = tdlc_connector.connect(region="ap-***", #填入正确地址，如ap-Singapore,ap-Shangh
    secret_id="*******",
    secret_key="*******",
    engine="your engine",#填入购买的引擎名称
    resource_group=None,
    engine_type=constants.EngineType.AUTO,
    result_style=constants.ResultStyles.LIST,
    download=True
    )
query = """
SELECT `sepal.length`, `sepal.width`,`petal.length`,`petal.width`,species FROM at_d
"""


#读取数据
iris = pd.read_sql(query, conn)
iris.head()
#划分数据集
X = iris[['petal.length', 'petal.width']].values
category_map = {
    'setosa': 0,
    'versicolor': 1,
    'virginica': 2
}

y= iris['species'].replace(category_map)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y)
print('Labels count in y:', np.bincount(y))
print('Labels count in y_train:', np.bincount(y_train))
print('Labels count in y_test:', np.bincount(y_test))

#数据归一化
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))

#逻辑回归进行分类，可视化分类结果
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=100.0, random_state=1, solver='lbfgs', multi_class='ovr')
lr.fit(X_train_std, y_train)
```

```
plot_decision_regions(X_combined_std, y_combined,
                      classifier=lr, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()

#查看模型准确率
y_pred = lr.predict(X_test_std)
print(accuracy_score(y_test, y_pred))
```

前往 MFlow 查看训练结果与注册模型

1. 选择机器学习功能。

2. 查看实验记录，并选择最优训练结果注册为模型。

StarRocks 直接查询 DLC 内部存储

最近更新时间：2024-10-31 11:13:30

StarRocks 是什么

StarRocks 是新一代极速全场景 MPP 数据库， 充分吸收了关系型 OLAP 数据库和分布式存储系统在大数据时代的优秀研究成果，在业界实践的基础上，进一步改进优化、升级架构，并增添了众多全新功能，形成了全新的企业级产品。StarRocks 致力于构建极速统一分析体验，满足企业用户的多种数据分析场景，支持多种数据模型(明细模型、聚合模型、更新模型)，多种导入方式（批量和实时），支持导入多达10000列的数据，可整合和接入多种现有系统(Spark、Flink、Hive、 ElasticSearch)。
在腾讯云弹性 MapReduce 产品中，我们提供了完全开箱即用的 StarRocks 服务，详细可参考 StarRocks简介。

StarRocks+DLC 湖仓一体查询加速

腾讯云数据湖 DLC 支持基于 EMR StarRocks 的湖仓一体查询加速模式，无需将 DLC 数据导入 StarRocks 或在 StarRocks 创建 DLC 外部表即可无缝分析 DLC 的数据源，并执行复杂的SQL查询。基于 StarRocks 的 MPP 向量化查询能力，提升了数据分析效率并降低了运维难度和成本。
接下来，本文为您介绍如何开启 DLC + （EMR）StarRocks 湖仓一体查询加速。

前提条件

1. 已购买弹性 MapReduce StarRocks 集群。
2. 已开通数据湖计算 DLC 服务。

注意：

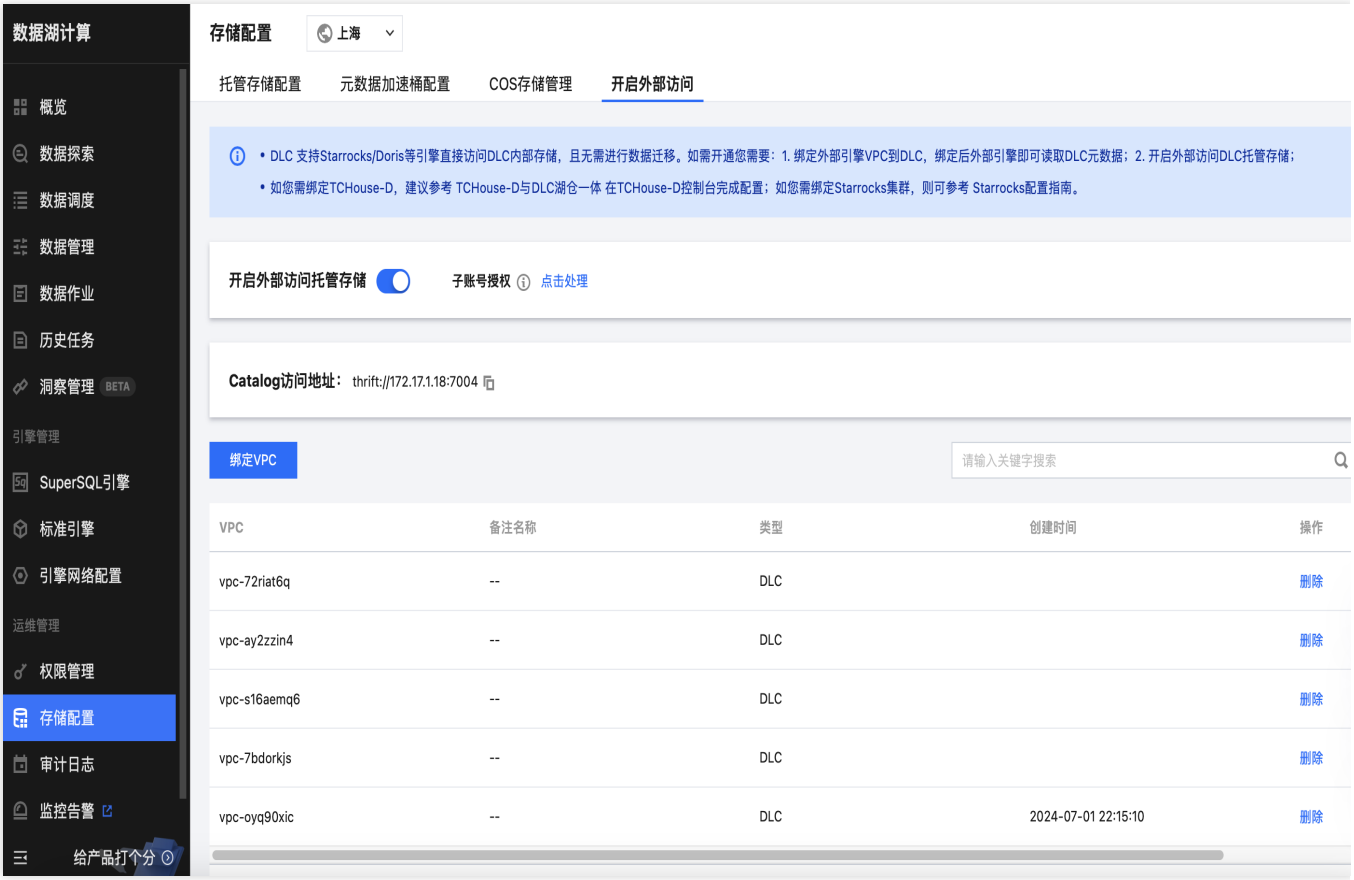
1. 当前暂不支持跨地域联邦 DLC，请合理规划环境，确保 EMR StarRocks、DLC 在相同地域；
2. 暂不支持查询2024年6月12日前创建的 DLC 存量原生表（DLC 存量原生表的存储路径为 lakefs://*** ，此格式暂不兼容）—— 可支持6月12日后创建的 DLC 新原生表（存储路径为cosn://***）及 DLC 所有外表；
3. StarRocks 仅支持查询 DLC 数据，不支持执行写和删除 DLC 数据的操作。

为开通 DLC+StarRocks 湖仓一体查询加速，您需要首先**开启 DLC 外部访问**，让 StarRocks 集群可访问到 DLC 内部托管存储的数据。随后在 StarRocks 集群中创建 **DLC External Catalog**，即可使用 StarRocks 计算引擎直接分析 DLC 存储数据。

开启 DLC 外部访问

进入 数据 数据湖计算 DLC 控制台，点击**存储配置**菜单，选择**开启外部访问**。

第一步：点击**“开启外部访问托管存储”**，开启后您同一腾讯云账号下的 EMR StarRocks 集群即可访问 DLC 内部托管存储。后续您在此通过点击开关，来开启或关闭 DLC 内部托管存储的外部访问服务。



开启外部访问的账号需要**有 DLC 的管理权限**，请使用主账号（或有 DLC 管理员权限的子账号）进行此操作。

注意：

1. 若您已有 DLC 的管理员权限，此步骤可忽略；

2. 若您使用的是子账号，且没有 DLC 管理员权限，可参考文档 子账号权限管理 让有 DLC 管理员权限的账号为您授权。

第二步：为确保您 EMR StarRocks 集群能正确访问到 DLC 的元数据 Catalog 服务，您需要将 EMR StarRocks 的 VPC 绑定到 DLC 网络中。

1. 点击“绑定VPC”，在对话框中类型选择 EMR StarRocks，并在 EMR 实例下拉列表选择需要绑定的 EMR StarRocks 集群实例 ID；
2. StarRocks 集群所在 VPC 将被自动填充，您可在备注名称中输入易于识别的别名；

第三步：完成 VPC 绑定后，您在 StarRocks 集群中即可通过 Catalog 访问地址显示的 URI 连接串，连接到 DLC 元数据服务，如：

URI 连接串示例：

Catalog 访问地址： thrift://172.17.1.18:7004

到此，您的 EMR StarRocks 集群已经可直接分析 DLC 内部托管数据。在开始分析前，您还需要在 StarRocks 中创建 DLC External Catalog。

创建 DLC External Catalog

登录 StarRocks 并在 StarRocks 中创建 DLC Catalog。关于External Catalog 详情，请参见 CREATE EXTERNAL CATALOG | Overview。

语法

```
CREATE EXTERNAL CATALOG dlc_iceberg_cos_catalog
PROPERTIES
(
    "type" = "iceberg",
        "iceberg.catalog.type" = "hive",
    "iceberg.catalog.hive.metastore.uris" = "thrift://169.254.0.171:8007",
    "aws.s3.endpoint" = "cos.ap-chongqing.myqcloud.com",
    "aws.s3.access_key" = "腾讯云secret id",
    "aws.s3.secret_key" = "腾讯云secret_key"
);
```

参数说明

参数	说明
type	数据源的类型，默认设置为 Iceberg。

iceberg.catalog.type	Iceberg 集群所使用的元数据服务的类型。设置为 hive。
iceberg.catalog.hive.metastore.uris	DLC 元数据的 URI。
aws.s3.endpoint	用于访问兼容 S3 协议的对象存储的 Endpoint, 腾讯云cos的格式为用于访问兼容 S3 协议的对象存储的 Endpoint, 腾讯云cos的格式为cos.<region>.myqcloud.com, <region>可选值为ap-beijing,ap-shanghai,ap-guangzhou 等。
aws.s3.access_key	腾讯云账号密钥中的 SecretID。
aws.s3.secret_key	腾讯云账号密钥中的 SecretKey。

注意：

1. 出于安全性考虑，在上述 SecretID 及 SecretKey 配置中，您需要使用主账号的 SecretID 及 SecretKey 才能正确访问 DLC 内部存储。具体可通过主账号登录腾讯云后，在访问管理控制台 >登录 - 腾讯云页面获取 SecretID 和 SecretKey；
2. 如果您需要使用子账号的 SecretID 及 SecretKey 访问 DLC 内部托管存储，可参考本文档最后一部分**使用子账号 SecretID/SecretKey 访问**。

示例

以下示例演示创建一个名为 DLC_catalog 的 DLC Catalog ：

```
CREATE EXTERNAL CATALOG dlc_hive_cos_catalog
PROPERTIES
(
    "type" = "Iceberg",
    "iceberg.catalog.type" = "hive",
    "hive.metastore.uris" = "thrift://169.254.0.171:8007",
    "aws.s3.endpoint" = "cos.ap-chongqing.myqcloud.com",
        "aws.s3.access_key" = "************************************",
    "aws.s3.secret_key" = "********************************"
)
```

您如需要通过 StarRocks 直接查询 DLC 托管存储中的 Hive 表，则您需要创建另一个 DLC Hive catalog，具体将 type 设置为 hive，示例如下：

```
CREATE EXTERNAL CATALOG dlc_hive_cos_catalog
PROPERTIES
(
    "type" = "hive",
    "iceberg.catalog.type" = "hive",
    "hive.metastore.uris" = "thrift://169.254.0.171:8007",
```

```
    "aws.s3.endpoint" = "cos.ap-chongqing.myqcloud.com",
        "aws.s3.access_key" = "**********************************",
    "aws.s3.secret_key" = "*******************************"
)
```

查询DLC数据

DLC准备表和数据

创建iceberg表示例如下：

```
CREATE TABLE test_sr_ofs.`customer`(
    `c_custkey` bigint,
    `c_name` string,
    `c_address` string,
    `c_nationkey` int,
    `c_phone` string,
    `c_acctbal` double,
    `c_mktsegment` string,
    `c_comment` string) using iceberg;
)
```

StarRocks中查询数据

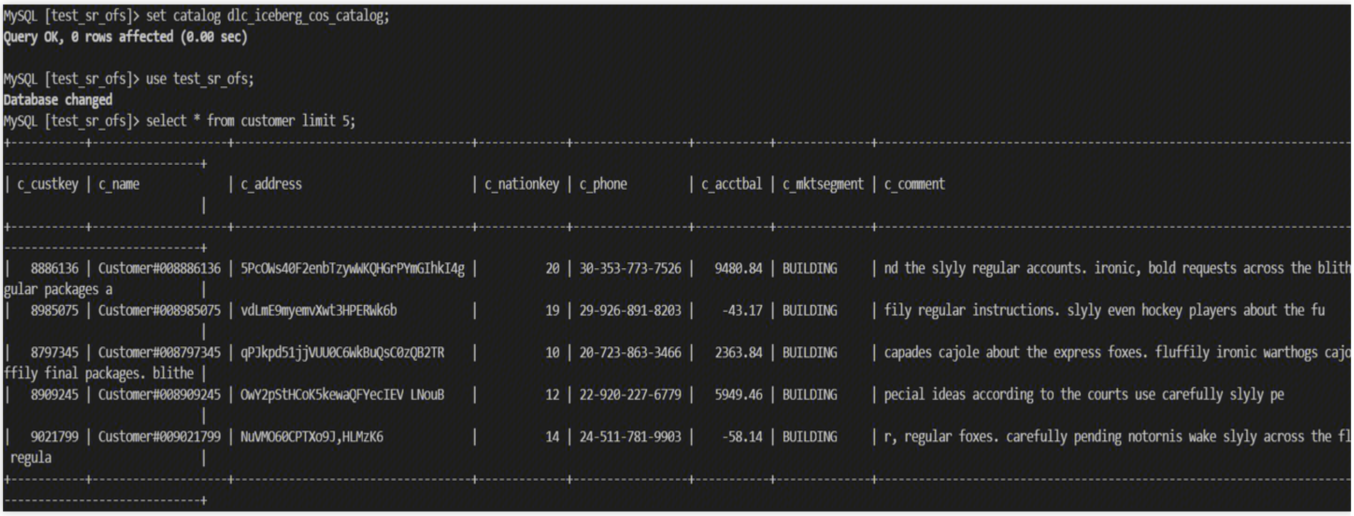
```
#登录StarRocks节点
mysql -h 172.30.0.xxx -P9030 -u root -p

#指定iceberg catalog
set catalog dlc_iceberg_cos_catalog;

#指定数据库
use test_sr_ofs;

#查询customer表数据
select * from customer limit 5;
```

查询结果如下：



```
MySQL [test_sr_ofs]> set catalog dlc_iceberg_cos_catalog;
Query OK, 0 rows affected (0.00 sec)

MySQL [test_sr_ofs]> use test_sr_ofs;
Database changed
MySQL [test_sr_ofs]> select * from customer limit 5;
+----------+--------------------+----------------------------------+-------------+-----------------+-----------+--------------+------------------------------------------------------------
---------------------------+
| c_custkey | c_name            | c_address                        | c_nationkey | c_phone         | c_acctbal | c_mktsegment | c_comment
                           |
+----------+--------------------+----------------------------------+-------------+-----------------+-----------+--------------+------------------------------------------------------------
---------------------------+
|   8886136 | Customer#008886136 | 5PcOWs40F2enbTzywWKQHGrPYmGIhkI4g |          20 | 30-353-773-7526 |   9480.84 | BUILDING     | nd the slyly regular accounts. ironic, bold requests across the blith
gular packages a           |
|   8985075 | Customer#008985075 | vdLmE9myemvXwt3HPERWk6b          |          19 | 29-926-891-8203 |    -43.17 | BUILDING     | fily regular instructions. slyly even hockey players about the fu
                           |
|   8797345 | Customer#008797345 | qPJkpd51jjVUU0C6WkBuQsC0zQB2TR   |          10 | 20-723-863-3466 |   2363.84 | BUILDING     | capades cajole about the express foxes. fluffily ironic warthogs cajo
ffily final packages. blithe |
|   8909245 | Customer#008909245 | OwY2pStHCoK5kewaQFYecIEV LNouB    |          12 | 22-920-227-6779 |   5949.46 | BUILDING     | pecial ideas according to the courts use carefully slyly pe
                           |
|   9021799 | Customer#009021799 | NuVMO60CPTXo9J,HLMzK6            |          14 | 24-511-781-9903 |    -58.14 | BUILDING     | r, regular foxes. carefully pending notornis wake slyly across the fl
 regula                    |
+----------+--------------------+----------------------------------+-------------+-----------------+-----------+--------------+------------------------------------------------------------
---------------------------+
```

使用子账号 SecretID/SecretKey 访问（可选）

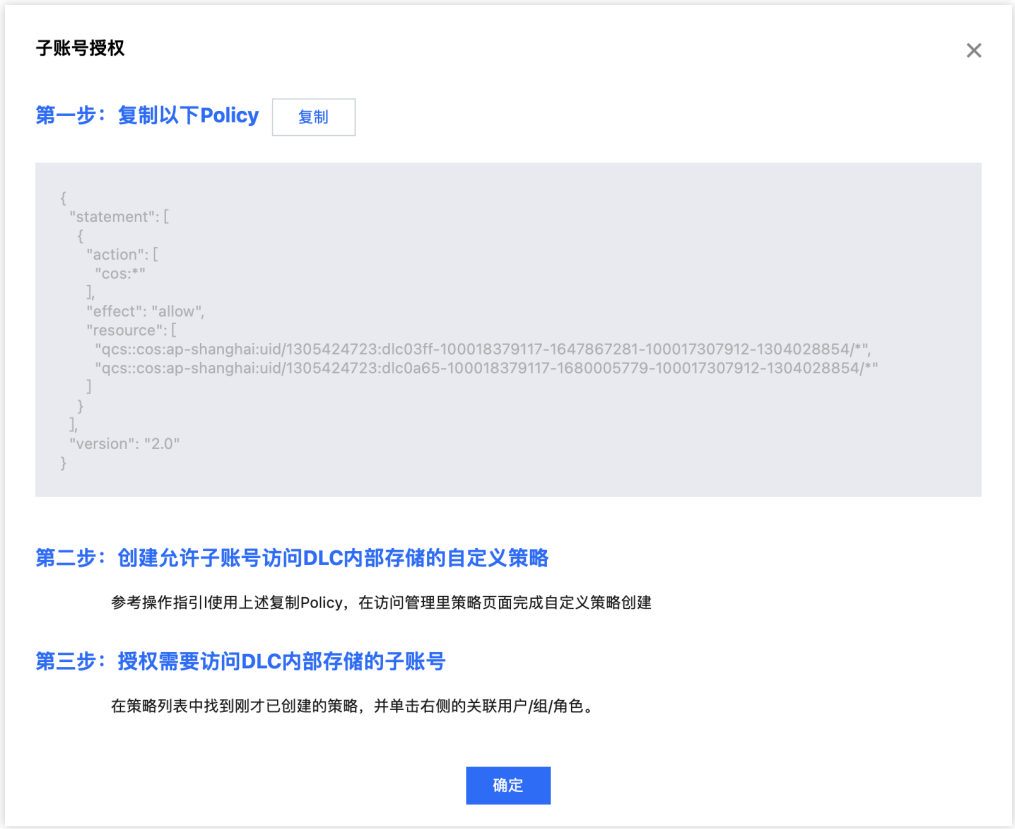
由于**数据安全考虑**，开启 DLC 外部访问后默认需要使用主账号 SecretID 及 SecretKey 在 EMR StarRocks 发起对 DLC 内部存储的访问。如因业务场景确需使用子账号 SecretID 及 SecretKey 访问，**需要您使用主账号在 CAM 完成自定义策略创建并绑定到对应子账号**。

原理解释

在您按照本文档上述指引开启 DLC 外部访问后，实质上系统授权了您的主账号具备访问 DLC 内部托管存储的权限。但出于数据安全考虑，您主账号名下其他子账号**默认不具备此访问权限**，需要您在腾讯云访问控制 CAM 中创建授权子账号具备访问能力的自定义策略，并绑定到需要访问 DLC 内部托管存储的子账号。具体步骤如下：

第一步：生成自定义策略

使用主账号登录腾讯云，在进入 数据湖计算 DLC 控制台，点击**存储配置**菜单，选择**开启外部访问**，在**子账号授权**中，点击“**点击处理**”。在弹出对话框**点击复制**，获得您需要创建的自定义 CAM 策略。



第二步：创建允许子账号访问 DLC 内部存储的自定义策略

1. 使用主账号登录访问管理 CAM 控制台的 策略 页面。
2. 选择**新建自定义策略 > 按策略语法创建**，随后选择空白模板，单击下一步。

说明：

主账号 给子账号授权数据访问权限，目前只能通过自定义策略授权，不支持通过预设策略授权。

3. 填写如下表单：

策略名称：自行定义一个不重复且有意义的策略名称，例如 cos-child-account。

备注：可选，自行编写。

策略内容：粘贴**第一步复制**的自定义策略，如：

```
{
  "statement": [
    {
      "action": [
        "cos:*"
      ],
      "effect": "allow",
      "resource": [
        "qcs::cos:ap-shanghai:uid/1305424723:dlc03ff-100018379117-1647867281-100017
        "qcs::cos:ap-shanghai:uid/1305424723:dlc0a65-100018379117-1680005779-100017
```

```
        ]
      }
    ],
    "version": "2.0"
  }
```

说明：

以上策略表示将主账号有权操作的 DLC 托管存储均授权给子账号 。其中，uid/1305424723 中的1305424723为主账号 A 的 APPID，dlc0a65-100018379117-1680005779-100017307912-1304028854/*为您有权操作的 DLC 内部托管存储。

4. 单击**完成**，完成策略的创建。

第三步：授权需要访问 DLC 内部存储的子账号

1 在**策略**列表中找到刚才已创建的策略，并单击右侧的**关联用户/组/角色**。

2.在弹窗中，勾选需要具备访问 DLC 内部托管存储权限的子账号，单击**确定**。

3.完成授权操作，即可使用子账号的 SecretID 及 SecretKey，访问您名下 DLC 内部托管存储。

Spark 计算成本优化实践

最近更新时间：2025-06-26 11:25:16

成本优化是一个持续不断的过程。由于大数据的动态性和不断变化的性质，企业用户成本优化的活动应该持续不断的进行。本文为您介绍在数据湖计算 DLC 中，基于 Spark 计算资源，如何进行成本优化的相关实践。您可以参考我们提供的使用场景，按需使用优化。

如何选择合适的计算资源付费方式

数据湖计算 DLC 支持引擎通过按量付费与包年包月两种付费模式进行购买。

资源描述	按量付费计算资源	包年包月计算资源	弹性计算资源
费用标准	0.35元/CU*时 表示使用1CU资源一个小时收费0.35元。按实际使用的资源的CU量计费	150元/CU*月 表示1CU资源一个月的费用为150元	0.35元/CU*时 表示使用1CU资源一个小时收费0.35元。按实际使用的资源的CU量计费
付费方式	后付费	预付费	后付费
使用特点	1. 用户可以灵活选择资源的使用时间：当用户不使用按量资源时，可以选择将资源挂起（释放），挂起后不再继续收费。 2. 按量资源在用户拉起集群运行任务时才会分配给用户，分配成功后由当前用户独占；**但受资源池影响，可能出现资源不足按量资源无法分配出来的情况。**	1. 资源在购买后就会分配给用户独占，不会出现资源无法分配的情况。 2. 随时可用。同时支持加购**弹性资源，并按量计费。**	1. 弹性计算资源是**基于购买了 Spark 包年包月集群的基础下**，额外开通按量计费资源。 2. 弹性资源能够在必要的时候加快任务的运行速度，减少整个系统的负载。同时在任务较少时，弹性资源自动释放，不再收费，有效降低成本。
推荐场景	1. POC 测试阶段 2. 每个月使用时长不超过18天	1. 正式的生产环境 2. 适合任务量大且稳定的数据计算场景 3. 使用时间超过一个月总时间的60%，使用包年包月会更划算	1. 正式的生产环境 2. 购买的包年包月计算资源运行任务，任务的完成时间不符合预期
是否支持付费方式切换	是	**否**	**否**

场景：由于包年包月计算资源不足，导致任务完成时间不达预期

某电商平台采购了 128CU 包年包月的计算资源去保障600个分析任务可以在当日完成运行与结果返回。
随着电商大促来临，数据量激增，他们发现这段时间已无法保障任务的完成时效。通过分析，发现问题在于目前购买的资源无法满足大促期间的数据处理需求，从而导致任务排队等待，进而延误进度。面对这一状况，企业希望可以有一个既能短期保障任务按时完成，又能把成本控制在合理范围内的方案。

推荐方案：

大促期间，在128CU包年包月的基础上，额外购买128CU弹性计算资源，根据任务负载情况，当包年包月的128CU都在使用时，触发弹性资源运行，提升运行效率。大促结束后，关闭弹性功能，有效控制成本投入。
任务量少：按量计费的弹性资源自动释放，不再收费，有效降低成本。
任务量多：按量计费的弹性资源能够在必要的时候加快任务的运行速度，用多少，费用收多少。

开通步骤：

1. 进入引擎列表页，确认需要配置弹性计算资源的包年包月引擎。
2. 在操作栏单击“更多”，选择规格配置。

3. 打开弹性计算资源开关，并选择弹性配置规格。

注意：

弹性计算规格大小不能超过包年包月的规格。

4. 大促结束后，恢复正常任务运行，可单击规格配置，关闭弹性集群功能。

如何合理规划计算资源分配

方式一：通过多个引擎进行计算资源分配

通过购买多个计算资源，分配给不同用户组或功能场景，每个计算资源独立运行任务。

优势	不同引擎之间资源完全隔离，由当前用户组独占。不同用户组之间的配置、管理、任务和故障都相互不影响。
适用场景	1. 多部门需要使用平台，但彼此之间没有业务重合，多为独立业务分析场景。 2. 对成本审计、安全运维有较高要求的企业。
局限	资源有可能会有空闲时间，利用无法最大化。如图用户组01主要做SQL分析，主要在早上9点到下午5点之间使用资源，其他时间资源处于空闲状态。 **解决方式：**调整计算资源为按量计费资源、调整资源分配方式。

方式二：一个引擎通过资源组进行资源分配

该模式下，所有用户组或功能场景使用同一个引擎，但每个用户使用配置不同的资源组来进行资源分配。

优势	能够实现计算资源利用最大化。
适用场景	1. 对成本控制有较高要求。 2. 需要配置的资源组数量不多，且多为线性启动任务，较少并发任务。 3. 各任务运行时间不会产生太多重合，且重合任务消耗计算量不超过资源总量。
局限	不同的资源组之间可能会形成资源竞争，如图三个用户均在使用的情况下，如果用户一占用了256CU资源，用户二也占用了256CU资源，此时引擎维度的所有资源均被占用，用户三会分配不到资源，导致用户三任务无法运行。 **解决方式：**增加计算资源、调整资源组配置与分配方式。

场景：通过合理规划任务的分时运行，实现资源利用最大化

某企业现需要购买计算资源给三个部门分配使用，他们分别需要进行：交互式 SQL 分析与 Spark 批作业任务。企业有较高的成本控制的要求，故希望销售团队可以出给出一个合理的购买方案建议：在资源利用最大的基础下，即能够保障资源可以合理分配给三个部门使用，且不能出现任务失败的情况。

推荐方案：

1. 针对交互式 SQL 任务，需要根据每天需要运行的任务个数和复杂度，评估交互式资源组的运行时间、最小资源数量和弹性情况。
该部门经过分析得出：在每天9点到5点跑交互式SQL，根据SQL任务最大的并发数和资源需求数，确定该资源组需要使用的最大规格128CU，可以配置资源使用为4CU~128CU。其中4CU为资源组资源最小值。
2. 针对作业任务，需要根据任务的运行时效去评估需要的资源。
如该企业，部门B有一个定时任务，每个小时跑一次，若要在当前小时运行结束，至少使用128CU资源才能保证任务按时完成，那么可以配置作业的资源为4cu~128CU。
部门C也有任务每天跑一次，只需要在凌晨运行，但必须在早上八点前运行完成，需要资源256CU。
这里发现：
部门 A 和 B 的任务时间有重叠，所以引擎最大资源需要配置：128 + 128 = 256CU
而部门 B 和部门 C 的任务时间也有重叠，为了保证任务按时完成，需要配置最大资源为：128 + 256 = 384CU
故销售根据每天运行的时间使用情况，确认给出：“128CU包年包月 + 128CU弹性计费”的资源购买建议。

如何进行成本追踪

方式一：通过分账实现成本追踪

数据湖计算 DLC 支持根据引擎维度进行标签分账，通过给引擎打标签，能够在费用中心实现标签查看分账。

如何所示：通过为引擎打上不同的标签后，标签 A 可以看到引擎1、2的费用统计，B 可以看到引擎3的费用，C 可以看到引擎2、3的费用统计。

说明：

如何为引擎打标签，详情请参见 引擎资源关联标签。

如何使用标签分账，详情请参见 分账标签。

方式二：通过任务的洞察功能实现成本追踪

通过进入数据湖计算 DLC 的历史任务实例页面，找到已经完成的任务，单击任务名称/ID，即可查看该任务的CU消耗*时。

说明：

1. CU消耗*时：含为参与计算所用 Spark Executor 每个 core 的 CPU 执行时长总和，单位小时。
2. 该资源消耗是该任务实际占用资源产生的消耗，没有包含资源拉起、资源组空闲等时间统计，故该值会远小于资源组的总消耗，无法和计费侧的数据完全一致。该消耗值推荐用于分析单个任务实际使用的资源量，用于单个任务的优化，或者进行用量大小等成本分析。

如何通过任务治理，提升任务完成时效，达到成本优化

数据湖计算 DLC 洞察管理 提供了一个可视化的直观界面，帮助您快速了解当前查询性能表现以及影响性能的潜在因素，并获取性能优化建议。详情请参见任务洞察。

适用场景	1. 对 Spark 引擎整体运行状况的洞察分析，例如：引擎下各任务运行时的资源抢占情况，引擎内资源使用情况，引擎执行时长，数据扫描大小，数据shuffle 大小等都有直观的展示与分析。 2. 自助排查分析任务运行情况，例如：可对众多任务按照耗时筛选排序，快速找到有问题的大任务，定位 Spark 任务运行缓慢或者失败的原因，如资源抢占，shuffle 异常，磁盘不足等情况，都有清晰的定位。
重点指标	1. 引擎执行时间：Spark 引擎执行的第一个 Task 时间（即任务第一次抢占 CPU 开始执行的时间）。 2. 引擎内执行耗时：反映真正用于计算所需的耗时，即从 Spark 任务第一个 Task 开始执行到任务结束之间的耗时。 3. 排队耗时：从任务提交到开始执行第一个 Spark Task 之间的耗时，其中耗时可能有：引擎第一次执行的冷启动耗时、配置任务并发上限导致的排队时间 、引擎内因资源打满导致的等待 executor 资源的耗时、资源组无法分配到资源导致的排队耗时。 4. 消耗 CU * 时：统计参与计算所用 Spark Executor 每个 core 的 CPU 执行时长总和，单位CU*小时。

场景1：通过任务问题自动分析，快速定位问题，提升任务运行时效

任务质量参差不齐，众多任务难以运维，导致集群资源利用率低下。

导致任务的运行慢的因素主要有：数据倾斜，shuffle 并发度不够，长尾任务拖慢整体执行时间等。通过数据湖计算 DLC 任务洞察 功能支持对任务运行过程中遇到的部分问题进行分析，并提出优化方案。在时间有限的情况下，可优先找到头部大任务，优化效果收益更好。

1. 例如按照引擎执行耗时排序（或者 消耗 CU * 时耗时排序）后，筛选有问题的任务。
2. 例如：Spark shuffle 阶段是影响任务执行速度和整体集群稳定关键因素，针对大 shuffle 数据量的任务进行定向优化收益会比较明显。具体操作：可以按任务 shuffle 大小排序，筛选 shuffle 异常任务（成功任务内部也可能会存在 shuffle 失败重试的情况），或者定向优化头部大 shuffle 任务对整体集群消耗收益都比较好。

可重点关注以下指标：

洞察类型	算法描述（正持续改进和新增算法）
资源抢占	sql 开始执行的 task 延迟时间＞stage 提交时间1分钟，或延迟时长超过总运行时长的20%（不同运行时长和数据量的任务，阈值公式会有动态调整）
shuffle 异常	stage 执行出现 shuffle 相关错误栈信息
慢 task	stage 中 task 时长 > stage 里其他 task 平均时长的2倍（不同运行时长和数据量的任务，阈值公式会有动态调整）
数据倾斜	task shuffle 数据 > task 平均 shuffle 数据大小的2倍 (不同运行时长和数据量的任务，阈值公式会有动态调整)
磁盘或内存不足	stage 执行错误栈信息中包含了 oom 或者 磁盘不足的信息 或者 cos 带宽限制报错
输出较多小文件	（该洞察类型的收集需 Spark 引擎内核升级至 2024.11.16 之后的版本） 参考列表中的指标 "输出小文件个数"，满足下述一个条件则判定为 "存在输出较多小文件" ： 1. 分区表，若某个分区写出的小文件超过 200 个 2. 非分区表，输出小文件总数超过 1000 个 3. 分区、非分区表写出文件超过 3000 个，平均文件大小小于 4MB

场景2：通过引擎洞察，快速分析资源抢占情况，合理安排任务运行数量，提升任务运行时效

执行慢并不等于计算本身慢，在集群资源有限的情况下，容易造成任务会互相争抢资源，通过洞察管理功能，参考引擎内执行耗时和排队耗时两个指标，找到互相影响的任务后，合理调整任务排队规划。

例如下图，任务提交，引擎不一定会开始执行，从任务提交到开始执行第一个 Spark Task 之间可能包含：引擎第一次执行的冷启动耗时、配置任务并发上限导致的排队耗时 、引擎内因资源打满导致的等待 executor 资源的耗时、生成和优化 Spark 执行计划耗时等。

当集群资源不足时，这种前期等待资源耗时会更加明显。

同时任务洞察也支持对资源抢占的自动发现，如下图。并且可以查看和该任务并发的其他任务是否在抢占资源。

DLC 原生表
DLC 原生表核心能力

最近更新时间：2024-07-31 17:34:35

概述

DLC 原生表（Iceberg）是基于 Iceberg 湖格式打造的性能强、易用性高、操作使用简单的表格式，用户可在该基础上完整数据探索，建设 Lakehouse 等应用。用户首次在使用 DLC 原生表（Iceberg）时需要按照如下5个主要步骤进行：

1. 开通 DLC 托管存储。
2. 购买引擎。
3. 创建数据库表。结合使用场景，选择创建 append 或者 upsert 表，并携带优化参数。
4. 配置数据优化。结合表类型，选择独立的优化引擎，配置优化选项。
5. 导入数据到 DLC 原生表。DLC 支持多种数据写入，如 insert into/merge into/upsert，支持多种导入方式，如 spark/presto/flink/inlong/oceanus。

Iceberg 原理解析

DLC 原生表（Iceberg）采用 Iceberg 表格式作为底层存储，在兼容开源 Iceberg 能力的基础上，还做了存算分离性能、易用性增强。

Iceberg 表格式通过划分数据文件和元数据文件管理用户的数据。

数据层（data layer）：由一系列 data file 组成，用于存放用户表的数据，data file 支持 parquet、avro、orc 格式，DLC 中默认为 parquet 格式。

由于 iceberg 的快照机制，用户在删除数据时，并不会立即将数据从存储上删除，而是写入新的 delete file，用于记录被删除的数据，根据使用的不同，delete file 分为 position delete file 和 equality delete file。

position delete 是位置删除文件，记录某个 data file 的某一行被删除的信息。

equality delete 为等值删除文件，记录某个 key 的值被删除，通常用在 upsert 写入场景。delete file 也属于 data file 的一种类型。

元数据层（metadata layer）：由一系列的 manifest、manifest list、metadata 文件组成，manifest file 包含一系列的 data file 的元信息，如文件路径、写入时间、min-max 值、统计值等信息。

manifest list 由 manifest file组成，通常一个 manifest list 包含一个快照的 manifest file。

metadata file 为 json 格式，包含一些列的 manifest list 文件信息和表的元信息，如表 schema、分区、所有快照等。每当表状态发生变化时，都会产生一个新的 metadata file 覆盖原有 metadata 文件，且该过程有 Iceberg 内核的原子性保证。

原生表使用场景

DLC 原生表（iceberg）作为 DLC lakehouse 主推格式，从使用场景上，可分为 Append 场景表和 Upsert 表，Append 场景表采用的V1格式，Upsert 表采用的V2格式。

Append 场景表：该场景表仅支持 Append，Overwrite，Merge into 方式写入。

Upsert 场景表：相比 Append，写入能力相比多一种 Upsert 写入模式。

原生表使用场景及特点汇总如下表描述。

表类型	使用场景及建议	特点
原生表（iceberg）	1. 用户有实时写入需求，包括 append/merge into/upsert 场景需求，不限于 inlong/oceans/自建 flink 实时写入。 2. 用户不想直接管理存储相关的运维，交给 DLC 存储托管。 3. 用户不想对 Iceberg 表格式进行运维，交给 DLC 进行调优和运维。 4. 希望使用 DLC 提供的自动数据优化能力，持续对数据进行优化。	1. Iceberg 表格式。 2. 使用前需要开通托管存储。 3. 数据存储存储在 DLC 提供的托管存储上。 4. 不需要指定 external、location 等信息。 5. 支持开启 DLC 智能数据优化。

为更好的管理和使用 DLC 原生表（Iceberg），您创建该类型的表时需要携带一些属性，这些属性参考如下。用户在创建表时可以携带上这些属性值，也可以修改表的属性值，详细的操作请参见 DLC 原生表操作配置。

属性值	含义	配置指导
format-version	iceberg 表版本，取值范围1、2，默认取值为1	如果用户写入场景有 upsert，该值必须设置为2
write.upsert.enabled	是否开启 upsert，取值为 true；不设置则为不开启	如果用户写入场景有 upsert，必须设置为 true
write.update.mode	更新模式	merge-on-read 指定为 MOR 表，缺省为 COW
write.merge.mode	merge 模式	merge-on-read 指定为 MOR 表，缺省为 COW
write.parquet.bloom-filter-enabled.column.{col}	开启 bloom，取值为 true 表示开启，缺省不开启	upsert 场景必须开启，需要根据上游的主键进行配置；如上游有多个主键，最多取前两个；开启后可提升 MOR 查询和小文件合并性能
write.distribution-mode	写实模式	建议取值为 hash，当取值为 hash 时，当数据写入时会自行进行 repartition，缺点是影响部分写入性能
write.metadata.delete-after-commit.enabled	开始 metadata 文件自动清理	强烈建议设置为 true，开启后 iceberg 在产生快照时会自动清理历史的 metadata 文件，可避免大量的 metadata 文件堆积

write.metadata.previous-versions-max	设置默认保留的 metadata 文件数量	默认值为100，在某些特殊的情况下，用户可适当调整该值，需要配合 write.metadata.delete-after-commit.enabled 一起使用
write.metadata.metrics.default	设置列 metrices 模型	必须取值为 full

原生表核心能力

存储托管

DLC 原生表（Iceberg）采用了存储数据托管模式，用户在使用原生表（Iceberg）时，需要先开通托管存储，将数据导入到 DLC 托管的存储空间。用户采用 DLC 存储托管，将获得如下收益。
数据更安全：Iceberg 表数据分为元数据和数据两个部分，一点某系文件被破坏，将导致整个表查询异常（相比于 Hive 可能是损坏的文件数据不能查询），存储托管在 DLC 可减少用户在不理解 Iceberg 的情况下对某些文件进行破坏。
性能：DLC 存储托管默认采用 chdfs 作为存储，相比于普通 COS，性能有较大的提升。
减少存储方面的运维：用户采用存储托管后，可不用再开通及运维对象存储，能减少存储的运维。
数据优化：DLC 元生表（Iceberg）采用存储托管模式，DLC 提供的数据优化能针对元生表持续优化。

ACID 事务

Iceberg 写入支持在单个操作中删除和新增，且不会部分对用户可见，从而提供原子性写入操作。
Iceberg 使用乐观并发锁来确保写入数据不会导致数据不一致，用户只能看到读视图中已经提交成功的数据。
Iceberg 使用快照机制和可序列化隔离级确保读取和写入是隔离的。
Iceberg 确保事务是持久化的，一旦移交成功就是永久性的。

写入

写入过程遵循乐观并发控制，写入者首先假设当前表版本在提交更新之前不会发生变更，对表数据进行更新/删除/新增，并创建新版本的元数据文件，之后尝试替换当前版本的元数据文件到新版本上来，但是在替换过程中，Iceberg 会检查当前的更新是否是基于当前版本的快照进行的，
如果不是则表示发生了写冲突，有其他写入者先更新了当前 metadata，此时写入必须基于当前的 metadata 版本从新更新，之后再次提交更新，整个提交替换过程由元数据锁保证操作的原子性。

读取

Iceberg 读取和写入是独立的过程，读者始终只能看到已经提交成功的快照，通过获取版本的 metadata 文件，获取的快照信息，从而读取当前表的数据，由于在未完成写入时，并不会更新 metadata 文件，从而确保始终从已经完成的操作中读取数据，无法从正在写入的操作中获取数据。

冲突参数配置

DLC 托管表（Iceberg）在写入并发变高时将会触发写入冲突，为降低冲突频率，用户可从如下方面对业务进行合理的调整。
进入合并的表结构设置，如分区，合理规划作业写入范围，减少任务写入时间，在一定程度上降低并发冲突概率。
作业进行一定程度的合并，减小写入并发量。
DLC 还支持一些列冲突并发重试的参数设置，在一定程度可提供重试操作的成功率，减小对业务的影响，参数含义及配置指导如下。

属性值	系统默认值	含义	配置指导
commit.retry.num-retries	4	提交失败后的重试次数	发生重试时，可尝试提大次数
commit.retry.min-wait-ms	100	重试前的最小等待时间，单位为毫秒	当时冲突十分频繁，如等待一段时间后依然冲突，可尝试调整该值，加大重试之间的间隔
commit.retry.max-wait-ms	60000（1 min）	重试前的最大等待时间，单位为毫秒	结合commit.retry.min-wait-ms一起调整使用
commit.retry.total-timeout-ms	1800000（30 min）	整个重试提交的超时时间	-

隐藏式分区

DLC 原生表（Iceberg）隐藏分区是将分区信息隐藏起来，开发人员只需要在建表的时候指定分区策略，Iceberg 会根据分区策略维护表字段与数据文件之间的逻辑关系，在写入和查询时无需关注分区布局，Iceberg 在写入数据是根据分区策略找到分区信息，并将其记录在元数据中，查询时也会更具元数据记录过滤到不需要扫描的文件。DLC 原生表（Iceberg）提供的分区策略如下表所示。

转换策略	描述	原始字段类型	转换后类型
identity	不转换	所有类型	与原类型一致
bucket[N, col]	hash分桶	int, long, decimal, date, time, timestamp, timestamptz, string, uuid, fixed, binary	int
truncate[col]	截取固定长度	int, long, decimal, string	与原类型一致
year	提取字段 year 信息	date, timestamp, timestamptz	int
month	提取字段 mouth 信息	date, timestamp, timestamptz	int

day	提取字段 day 信息	date, timestamp, timestamptz	int
hour	提取字段 hour 信息	timestamp, timestamptz	int

元数据查询和存储过程

DLC 原生表（Iceberg）可调用存储过程语句查询各类型表信息，如文件合并、快照过期等，如下表格提供部分常用的查询方法。

场景	CALL 语句	执行引擎
查询 history	select * from `DataLakeCatalog` . `db` . `sample$history`	DLC spark SQL 引擎、presto 引擎
查询快照	select * from `DataLakeCatalog` . `db` . `sample$snapshots`	DLC spark SQL 引擎、presto 引擎
查询 data 文件	select * from `DataLakeCatalog` . `db` . `sample$files`	DLC spark SQL 引擎、presto 引擎
查询 manifests	select * from `DataLakeCatalog` . `db` . `sample$manifests`	DLC spark SQL 引擎、presto 引擎
查询分区	select * from `DataLakeCatalog` . `db` . `sample$partitions`	DLC spark SQL 引擎、presto 引擎
回滚指定快照	CALL DataLakeCatalog. `system` .rollback_to_snapshot('db.sample', 1)	DLC spark SQL 引擎
回滚到某个时间点	CALL DataLakeCatalog. `system` .rollback_to_timestamp('db.sample', TIMESTAMP '2021-06-30 00:00:00.000')	DLC spark SQL 引擎
设置当前快照	CALL DataLakeCatalog. `system` .set_current_snapshot('db.sample', 1)	DLC spark SQL 引擎
合并文件	CALL DataLakeCatalog. `system` .rewrite_data_files(table => 'db.sample', strategy => 'sort', sort_order => 'id DESC NULLS LAST,name ASC NULLS FIRST')	DLC spark SQL 引擎
快照过期	CALL DataLakeCatalog. `system` .expire_snapshots('db.sample', TIMESTAMP '2021-06-30 00:00:00.000', 100)	DLC spark SQL 引擎
移除孤立文件	CALL DataLakeCatalog. `system` .remove_orphan_files(table => 'db.sample', dry_run => true)	DLC spark SQL 引擎
重新元数据	CALL DataLakeCatalog. `system` .rewrite_manifests('db.sample')	DLC spark SQL 引

		擎

数据优化

优化策略

DLC 原生表（Iceberg）提供的具备继承关系优化策略，用户可将策略配置在数据目录、数据库和数据表上。具体配置操作请参见 开启数据优化。

数据目录配置优化策略：该数据目录下的所有库下的所有的原生表（Iceberg）默认复用该数据目录优化策略。

数据库配置优化策略：该数据库下的所有原生表（Iceberg）默认复用该数据库优化策略。

数据表配置优化策略：该配置仅针对配置的原生表（Iceberg）生效。

用户通过上面的组合配置，可实现针对某库某表定制化优化策略，或者某些表关闭策略。

DLC 针对优化策略还提供高级参数配置，如用户对Iceberg熟悉可根据实际场景定制化高级参数，如下图所示。

DLC 针对高级参数设置了默认值。DLC 会将文件尽可能合并到128M大小，快照过期时间为2天，保留过期的5个快照，快照过期和清理孤立文件的执行周期分别为600分钟和1440分钟。

针对 upsert 写入场景，DLC 默认还提供合并阈值，该部分参数由 DLC 提供，在超过5min的时间新写入的数据满足其中一个条件将会触发小文件合并，如表所示。

参数	含义	取值
AddDataFileSize	写入新增数据文件数量	20
AddDeleteFileSize	写入新增Delete文件数据量	20
AddPositionDeletes	写入新增Position Deletes记录数量	1000
AddEqualityDeletes	写入新增Equality Deletes记录数量	1000

优化引擎

DLC 数据优化通过执行存储过程完成对数据的优化，因此需要数据引擎用于执行存储过程。当前 DLC 支持使用 Spark SQL 引擎作为优化引擎，在使用时需注意如下几点：

数据优化的 Spark SQL 引擎与业务引擎分开使用，可避免数据优化任务与业务任务相互抢占资源，导致任务大量排队和业务受阻。
生产场景建议优化资源64CU起，除非量特性表，如果小于10张表且单表数据量超过2G，建议资源开启弹性，防止突发流量，建议采用包年包月集群，防止提交任务时集群不可用导致优化任务失败。

参数定义

数据库、数据表数据优化参数设置在库表属性上，用户可以通过创建库、表时携带数据优化参数（DLC 原生表提供的可视化创建库表用户可配置数据优化）；用户也可通过 ALTER DATABASE/TABLE 对表数据进行表更，修改数据优化参数详细的操作请参见 DLC 原生表操作配置。

属性值	含义	默认值	取值说明
smart-optimizer.inherit	是否继承上一级策略	default	none：不继承；default：继承
smart-optimizer.written.enable	是否开启写入优化	disable	disable：不开启；enable：开启。默认不开启
smart-optimizer.written.advance.compact-enable	（可选）写入优化高级参数，是否开始小文件合并	enable	disable：不开启；enable：开启。
smart-optimizer.written.advance.delete-enable	（可选）写入优化高级参数，是否开始数据清理	enable	disable：不开启；enable：开启。
smart-optimizer.written.advance.min-input-files	（可选）合并最小输入文件数量	5	当某个表或分区下的文件数目超过最小文件个数时，平台会自动检查并启动文件优化合并。文件优化合并能有效提高分析查询性能。最小文件个数取值较大时，资源负载越高，最小文件个数取值较小时，执行更灵活，任务会更频繁。建议取值为5。
smart-optimizer.written.advance.target-file-size-bytes	（可选）合并目标大小	134217728 (128 MB)	文件优化合并时，会尽可能将文件合并成目标大小，建议取值128M。
smart-	（可选）	2	快照存在时间超过该值时，平台会将该

optimizer.written.advance.before-days	快照过期时间，单位天		快照标记为过期的快照。快照过期时间取值越长，快照清理的速度越慢，占用存储空间越多。
smart-optimizer.written.advance.retain-last	（可选）保留过期快照数量	5	超过保留个数的过期快照将会被清理。保留的过期快照个数越多，存储空间占用越多。建议取值为5。
smart-optimizer.written.advance.expired-snapshots-interval-min	（可选）快照过期执行周期	600（10 hour）	平台会周期性扫描快照并过期快照。执行周期越短，快照的过期会更灵敏，但是可能消耗更多资源。
smart-optimizer.written.advance.remove-orphan-interval-min	（可选）移除孤立文件执行周期	1440（24 hour）	平台会周期性扫描并清理孤立文件。执行周期越短，清理孤立文件会更灵敏，但是可能消耗更多资源。

优化类型

当前 DLC 提供写入优化和数据清理两种类型，写入优化对用户写入的小文件进行合并更大的文件，从而提供查询效率；数据清理则是清理历史过期快照的存储空间，节约存储成本。

写入优化

小文件合并：将业务侧写入的小文件合并为更大的文件，提升文件查询效率；处理写入的deletes文件和data文件合并，提升MOR查询效率。

数据清理

快照过期：执行删除过期的快照信息，释放历史数据占据的存储空间。

移除孤立文件：执行移除孤立文件，释放无效文件占据的存储空间。

根据用户的使用场景，在优化类型上有一定差异，如下所示。

优化类型	建议开启场景
写入优化	upsert 写场景：必须开启 merge into 写场景：必须开启 append 写入场景：按需开启
数据清理	upsert 写场景：必须开启 merge into 写入场景：必须开启 append 写入场景：建议开启，并结合高级参数及历史数据回溯需求配置合理的过期删除时间

DLC 的写入优化不仅完成小文件的合并，还可以提供手动构建索引，用户需要提供索引的字段及规则，之后 DLC 将产生对应的存储过程执行语句，从而完成索引的构建。该能在 upsert 场景和结合小文件合并同时进行，完成小文件合并的时候即可完成索引构建，大大提高索引构建能力。

该功能目前处于测试阶段，如需要使用，可 联系我们 进行配置。

优化任务

DLC 优化任务产生有时间和事件两种方式。

时间触发

时间触发是优化高级参数配置的执行时间，周期性地触发检查是否需要优化，如对应治理项满足条件后，将会产生对应的治理任务。当前时间触发的周期至少是60min，通常用在清理快照和移除孤立文件。
时间触发针对小文件合并类型的优化任务仍然有效，当时触发周期默认为60min。
V1表（需后端开启）情况，每60min触发一次小文件合并。
V2表情况，防止表写入慢，长时间达不到事件触发条件，当时间触发V2进行小文件合并时，需要满足上一次小文件合并时间间隔超过1小时。
当快照过期和移除孤立文件任务执行失败或者超时时，在下一个检查周期会再次执行，检查周期为60min。

事件触发

事件触发发生了表 Upsert 写入场景，主要是 DLC 数据优化服务后台会监控用户表数据的 Upsert 表写入的情况，当写的达到对应的条件时，触发产生治理任务。事件触发用在小文件并场景，特别是 flink upsert 实时写入场景，数据写入快，频繁产生小文件合并任务。
如数据文件阈值20，deletes 文件阈值20，则写入20个文件或者20个 deletes 文件，并同时满足相同任务类型之间的产生的间隔默认最小时间5min时，就会触发产生小文件合并。

生命周期

DLC 原生表的生命周期（Lifecycle），指表（分区）数据从最后一次更新的时间算起，在经过指定的时间后没有变动，则此表（分区）将被自动回收。DLC 元数据表的生命周期执行时，只是产生新的快覆盖过期的数据，并不会立即将数据从存储空间上移除，数据真正从存储上移除需要依赖于元数据表数据清理（快照过期和移除孤立文件），因此生命周期需要和数据清理一起使用。
注意：
生命周期目前处于邀测阶段，如需要开通，欢迎 联系我们。
生命周期移除分区时只是从当前快照中逻辑移除该分区，但是被移除的文件并不会立即从存储系统中删除，被移除的文件只有等到快照过期才会从存储系统上删除。

参数定义

数据库、数据表生命周期参数设置在库表属性上，用户可以通过创建库、表时携带生命周期参数（DLC 原生表提供的可视化创建库表用户可配置生命周期）；用户也可通过 ALTER DATABASE/TABLE 对表数据进行表更改，修改生命周期参数详细的操作请参见 DLC 原生表操作配置。

属性值	含义	默认值	取值说明
smart-optimizer.lifecycle.enable	是否开启生命周期	disable	disable：不开启；enable：开启。默认不开启
smart-	生命周期实现周	30	当 smart-optimizer.lifecycle.enable 取值为

optimizer.lifecycle.expiration	期，单位：天		enable 时生效，需大于1

结合 WeData 管理原生表生命周期

如果用户分区表的按照天分区，如分区值为yyyy-MM-dd或者yyyyMMdd的分区值，可配合WeData完成数据生命周期管理。

数据导入

DLC 原生表（Iceberg）支持多种方式的数据导入，根据数据源不同，可参考如下方式进行导入。

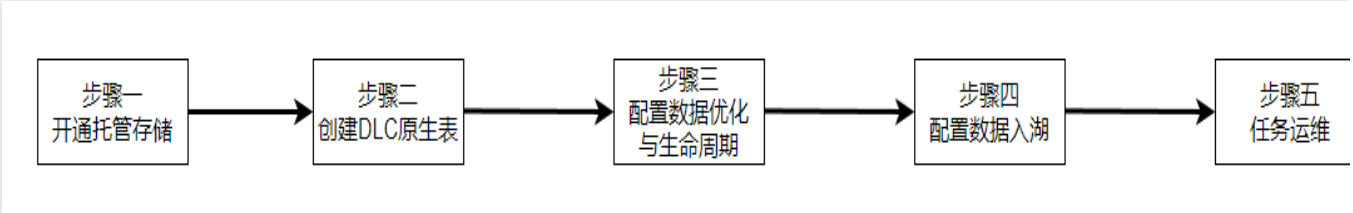
数据位置	导入建议
数据在用户自己的 COS 桶上	通过在 DLC 建立外部表，之后通过 Insert into/overwrite 的方式导入
数据在用户本地（或者其他执行机上）	用户需要将数据上传到用户自己的 COS 桶上，之后建立 DLC 外部表，通过 insert into/overwrite 的方式导入
数据在用户 mysql	用户可通过 flink/inlong/oceans 方式将数据导入，详细的数据入湖操作方式请参见 DLC原生表（Iceberg）入湖实践。
数据在用户自建 hive 上	用户通过建立联邦 hive 数据目录，之后通过 insert into/overwrite 的方式导入

DLC 原生表操作配置

最近更新时间：2024-07-31 17:34:51

概述

用户在使用 DLC 原生表（Iceberg）时，可以参考如下流程进行原生表创建和完成相关的配置。



步骤一：开启托管存储

说明：

托管存储需要 DLC 管理员开启。

开启托管存储需要在控制台上操作。详情请参见 托管存储配置。如果使用元数据加速桶，需注意权限配置，详情请参见 元数据加速桶的绑定。需注意共享引擎无法访问元数据加速桶。

步骤二：创建 DLC 原生表

创建原生表有两种方式。

1. 通过控制台界面可视化建表。
2. 通过 SQL 建表。

说明：

创建 DLC 原生表之前，需要先创建数据库。

通过控制台界面建表

DLC 提供数据管理模块进行建表，具体操作请参见数据管理。

通过 SQL 建表

SQL 建表时用户自己编写 CREATE TABLE SQL 语句进行创建，DLC 原生表（Iceberg）创建不需要指定表描述，不需要指定 location，不需要指定表格式。但根据使用场景，需要自行补充一些高级参数，参数通过 TBLPROPERTIES 的方式带入。

如果您在创建表的时候没有参数，或者要修改某些属性值，可通过 alter table set tblproperties 的方式进行修改，执行 alter table 修改之后，重启上游的导入任务即可完成属性值修改或者增加。

以下提供 Append 场景和 Upsert 场景的典型建表语句，用户在使用时可在该语句的基础上结合实际情况进行调整。

Append 场景建表

```
CREATE TABLE IF NOT EXISTS `DataLakeCatalog`.`axitest`.`append_case` (`id` int, `n
PARTITIONED BY (`pt`)
TBLPROPERTIES (
    'format-version' = '1',
    'write.upsert.enabled' = 'false',
    'write.distribution-mode' = 'hash',
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.metrics.default' = 'full',
    'smart-optimizer.inherit' = 'default'
);
```

Upsert 场景建表

Upsert 场景建表需要指定 version为2，设置 write.upsert.enabled 属性为 true，且要根据 upsert 的键值设置 bloom，如果用户有多个主键，一般取前两个键值设置 bloom。如果 upsert 表为非分区场景，且 upsert 更新频繁，数据量大，可根据主键做一定的分桶打散。

非分区表及分区表样例参考如下。

```
// 分区表
CREATE TABLE IF NOT EXISTS `DataLakeCatalog`.`axitest`.`upsert_case` (`id` int, `na
PARTITIONED BY (bucket(4, `id`))
TBLPROPERTIES (
    'format-version' = '2',
    'write.upsert.enabled' = 'true',
    'write.update.mode' = 'merge-on-read',
    'write.merge.mode' = 'merge-on-read',
    'write.parquet.bloom-filter-enabled.column.id' = 'true',
    'dlc.ao.data.govern.sorted.keys' = 'id',
    'write.distribution-mode' = 'hash',
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.metrics.default' = 'full',
    'smart-optimizer.inherit' = 'default'
);
```

```
// 非分区表
CREATE TABLE IF NOT EXISTS `DataLakeCatalog`.`axitest`.`upsert_case` (`id` int, `na
TBLPROPERTIES (
    'format-version' = '2',
    'write.upsert.enabled' = 'true',
    'write.update.mode' = 'merge-on-read',
    'write.merge.mode' = 'merge-on-read',
    'write.parquet.bloom-filter-enabled.column.id' = 'true',
    'dlc.ao.data.govern.sorted.keys' = 'id',
    'write.distribution-mode' = 'hash',
    'write.metadata.delete-after-commit.enabled' = 'true',
    'write.metadata.previous-versions-max' = '100',
    'write.metadata.metrics.default' = 'full',
    'smart-optimizer.inherit' = 'default'
);
```

修改表属性

如果用户在创建表的时候没有携带相关属性值，可通过 alter table 将相关属性值进行修改、添加和移除，如下所示。如涉及到表属性值的变更都可以通过改方式。特别的 Iceberg format-version 字段不能修改，另外如果用户表已经有 inlong/oceans/flink 实时导入，修改后需要重启上游导入业务。

```
// 修改冲突重试次数为10

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('commit.ret
```

```
// 取消name字段bloom设置

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` UNSET TBLPROPERTIES('write.pa
```

步骤三：数据优化与生命周期配置

数据优化与生命周期配置有两种方式。
1. 通过控制台界面可视化配置。
2. 通过 SQL 进行配置。

通过控制台界面配置

DLC 提供数据管理模块进行配置，具体操作请参见 开启数据优化。

通过 SQL 配置

DLC 定义了详细的属性用于管理数据优化与生命周期，您可以结合业务特点灵活配置数据管理与生命周期，详细的数据优化和生命周期配置值请参见 开启数据优化。

配置数据库

数据库的数据优化和生命周期可通过变更 DBPROPERTIES 进行，如下所示。

```
// 针对my_database表开启写入优化，且不继承数据目录策略

ALTER DATABASE DataLakeCatalog.my_database SET  DBPROPERTIES ('smart-optimizer.inhe
```

```
// 设置my_database继承数据目录策略

ALTER DATABASE DataLakeCatalog.my_database SET  DBPROPERTIES ('smart-optimizer.inhe
```

```
// 针对my_database表关闭生命周期，且不继承数据目录策略

ALTER DATABASE DataLakeCatalog.my_database SET  DBPROPERTIES ('smart-optimizer.inhe
```

配置数据表

数据表的数据优化和生命周期通过变更 TBLPROPERTIES 进行，如下所示。

```
// 针对upsert_cast表关闭写入优化，且不继承数据库策略

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('smart-opti
```

```
// 设置upsert_cast表继承数据库策略

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('smart-opti
```

```
// 针对upsert_cast表开启生命周期并设置生命周期时间为7天，且不继承数据库策略

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('smart-opti
```

步骤四：数据入湖到原生表

DLC 原生表支持多种多种方式的数据写入，结合您的数据写入方式，具体操作请参见 DLC 原生表入湖实践。

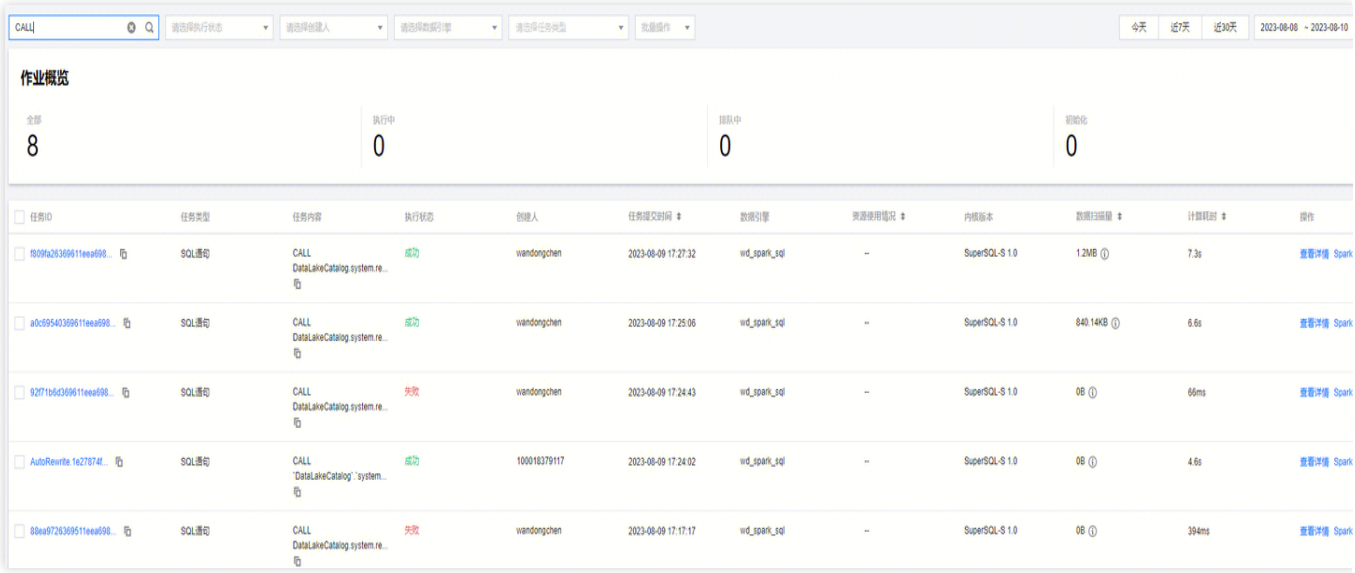
步骤五：查看数据优化任务

您可以在 DLC 控制台**数据运维**菜单，进入**历史任务**页面查看数据治理任务。可以"CALL"、"Auto"、库名称和表名称等关键字查询任务。

说明：

查看系统数据优化任务的用户需要具备 DLC 管理员权限。

任务 ID 为 Auto 开头的任务都是自动产生的数据优化任务。如下图所示。



您也可以点击**查看详情**，查询任务的基本信息和运行结果。

DLC 原生表入湖实践

最近更新时间：2024-07-31 17:35:06

应用场景

CDC（Change Data Capture）是变更数据捕获的缩写，可以将源数据库中的增量变更近似实时同步到其他数据库或应用程序。DLC 支持通过 CDC 技术将源数据库的增量变更同步到 DLC 原生表，完成源数据入湖。

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，详细配置请参考开启数据优化。

InLong 数据入湖

通过 DataInLong 可将源数据同步到 DLC。

Oceanus 流计算数据入湖

通过 Oceanus 可将源数据同步到 DLC。

自建 Flink 数据入湖

通过 Flink 可将源数据同步到 DLC。本示例展示将源 Kafka 的数据同步到 DLC，完成数据入湖。

环境准备

依赖集群：Kafka 2.4.x，Flink 1.15.x, Hadoop3.x。
Kafka、Flink 集群建议购买 EMR 集群。

整体操作过程

详细操作流程可参考如下图：

步骤1：上传依赖 Jar ：上传同步所需的 Kakfa、DLC 连接 Jar 包和 Hadoop 相关依赖 Jar。

步骤2：创建 Kafka Topic：创建 Kafka 生产消费的 Topic。

步骤3：DLC 新建目标表：DLC 数据管理新建目标表。

步骤4：提交任务：Flink 集群下提交同步任务。

步骤5：发送消息数据和查询同步结果：Kafka 集群发送消息数据和 DLC 上查看数据同步结果。

步骤1：上传依赖 Jar

1. 下载依赖 Jar

相关依赖 Jar 建议上传与Flink对应版本的Jar，例如 Flink 为 Flink1.15.x，则建议下载 flink-sql-connect-kafka-1.15.x.jar。相关文件参考附件。

Kafka 相关依赖：flink-sql-connect-kafka-1.15.4.jar

DLC 相关依赖：sort-connector-iceberg-dlc-1.6.0.jar

Hadoop3.x 相关依赖：api-util-1.0.0-M20.jar、guava-27.0-jre.jar、hadoop-mapreduce-client-core-3.2.2.jar。

2. 登录 Flink 集群，将准备好的 Jar 上传到 flink/ib 目录下。

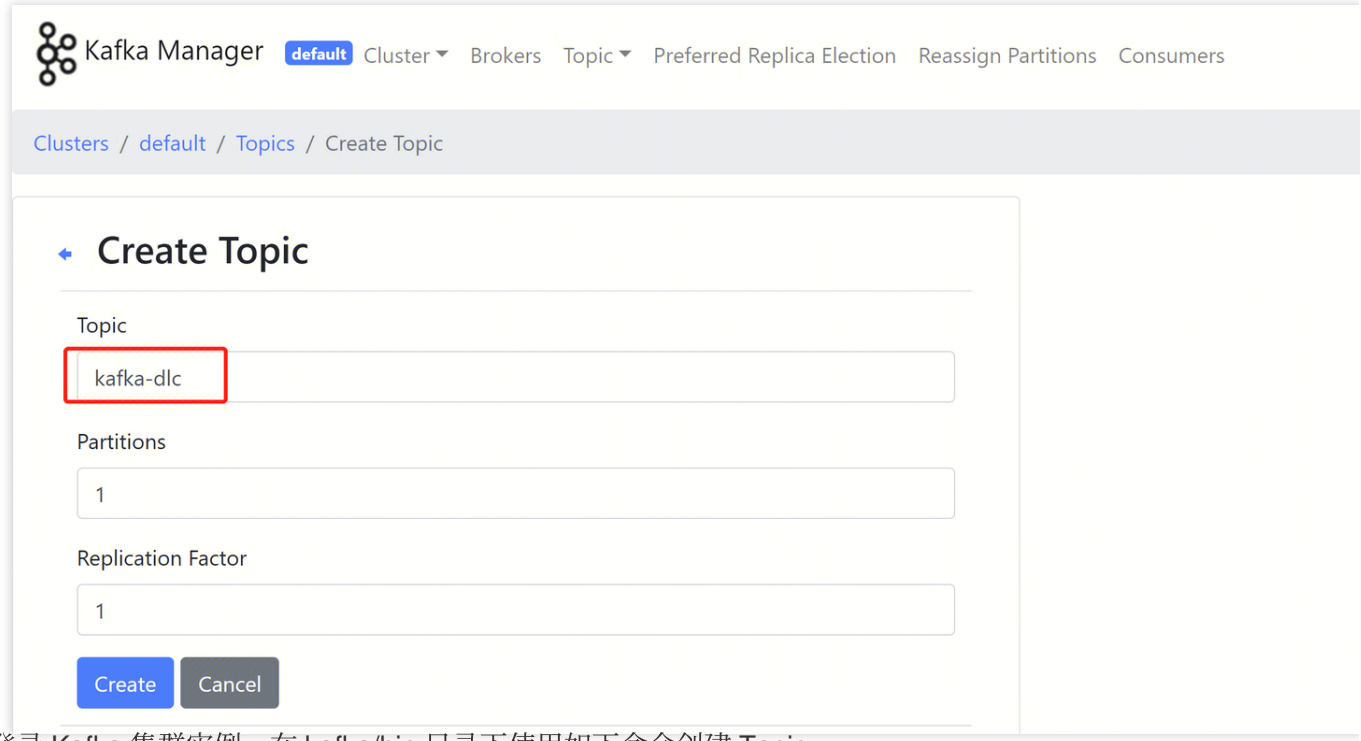
步骤2：创建 Kafka Topic

登录 Kafka Manager，单击 **default 集群**，单击 **Topic > Create**。

Topic 名称：本示例输入为 kafka_dlc

分区数：1

副本数：1



或者登录 Kafka 集群实例，在 kafka/bin 目录下使用如下命令创建 Topic。

```
./kafka-topics.sh --bootstrap-server ip:port --create  --topic kafka-dlc
```

步骤3：DLC 新建目标表

新建目标表详情可参考 DLC 原生表操作配置。

步骤4：提交任务

Flink 同步数据的方式有2种，Flink SQL写入模式 和 Flink Stream API，以下会介绍2种同步方式。
提交任务前，需要新建保存 checkpoint 数据的目录，通过如下命令新建数据目录。
新建 hdfs /flink/checkpoints 目录：

```
hadoop fs -mkdir /flink
hadoop fs -mkdir /flink/checkpoints
```

Flink SQL 同步模式

1. 通过 IntelliJ IDEA 新建一个名称为“flink-demo”的 Maven 项目。
2. 在 pom 中添加相关依赖，依赖详情请参考 完整样例代码参考 > 示例1。
3. Java 同步代码：核心代码如下步骤展示，详细代码请参考 完整样例代码参考 > 示例2。

创建执行环境和配置 checkpoint：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1);
env.enableCheckpointing(60000);
env.getCheckpointConfig().setCheckpointStorage("hdfs:///flink/checkpoints");
```

```
env.getCheckpointConfig().setCheckpointTimeout(60000);
env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
env.getCheckpointConfig().enableExternalizedCheckpoints(CheckpointConfig.Externaliz
```

执行 Source SQL：

```
tEnv.executeSql(sourceSql);
```

执行 同步 SQL：

```
tEnv.executeSql(sql)
```

4. 通过 IntelliJ IDEA 对 flink-demo 项目编译打包，在项目 target 文件夹下生成 JAR 包 flink-demo-1.0-SNAPSHOT.jar。
5. 登录 Flink 集群其中的一个实例，上传 flink-demo-1.0-SNAPSHOT.jar 到 /data/jars/ 目录（没有目录则新建）。
6. 登录 Flink 集群其中的一个实例，在 flink/bin 目录下执行如下命提交同步任务。

```
./flink run  --class com.tencent.dlc.iceberg.flink.AppendIceberg
/data/jars/flink-demo-1.0-SNAPSHOT.jar
```

Flink Stream API 同步模式

1. 通过 IntelliJ IDEA 新建一个名称为“flink-demo”的 Maven 项目。
2. 在 pom 中添加相关依赖：完整样例代码参考> 示例3。
3. Java 核心代码如下步骤展示，详细代码请参考 完整样例代码参考 > 示例4。

创建执行环境 StreamTableEnvironment，配置 checkpoint：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1);
env.enableCheckpointing(60000);
env.getCheckpointConfig().setCheckpointStorage("hdfs:///data/checkpoints");
env.getCheckpointConfig().setCheckpointTimeout(60000);
env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
env.getCheckpointConfig().enableExternalizedCheckpoints(CheckpointConfig.Externaliz
```

获取 Kafka 输入流：

```
KafkaToDLC dlcSink = new KafkaToDLC();
DataStream<RowData> dataStreamSource = dlcSink.buildInputStream(env);
```

配置 Sink：

```
FlinkSink.forRowData(dataStreamSource)
          .table(table)
          .tableLoader(tableLoader)
          .equalityFieldColumns(equalityColumns)
```

```
            .metric(params.get(INLONG_METRIC.key()), params.get(INLONG_AUDIT.key()))
            .action(actionsProvider)
            .tableOptions(Configuration.fromMap(options))
            //默认为false,追加数据。如果设置为true 就是覆盖数据
            .overwrite(false)
            .append();
```

执行同步 SQL：

```
env.execute("DataStream Api Write Data To Iceberg");
```

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

5. 登录 Flink 集群其中的一个实例，上传 flink-demo-1.0-SNAPSHOT.jar 到 /data/jars/ 目录（没有目录则新建）。

6. 登录 Flink 集群其中的一个实例，在 flink/bin 目录下执行如下命令提交任务。

```
./flink run  --class com.tencent.dlc.iceberg.flink.AppendIceberg
/data/jars/flink-demo-1.0-SNAPSHOT.jar
```

步骤5：发送消息数据和查询同步结果

1. 登录 Kafka 集群实例，在 kafka/bin 目录 用如下命令，发送消息数据。

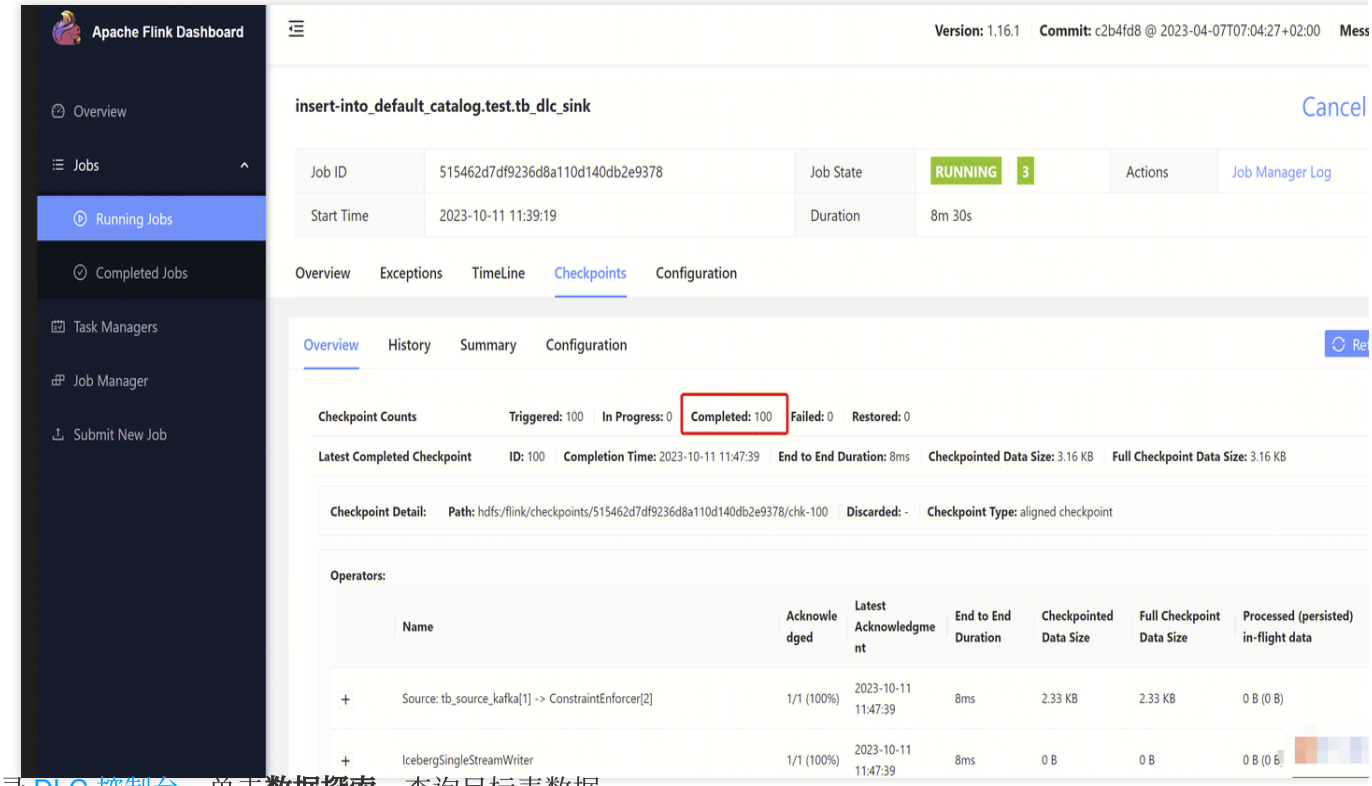
```
./kafka-console-producer.sh --broker-list 122.152.227.141:9092 --topic kafka-
dlc
```

数据信息如下：

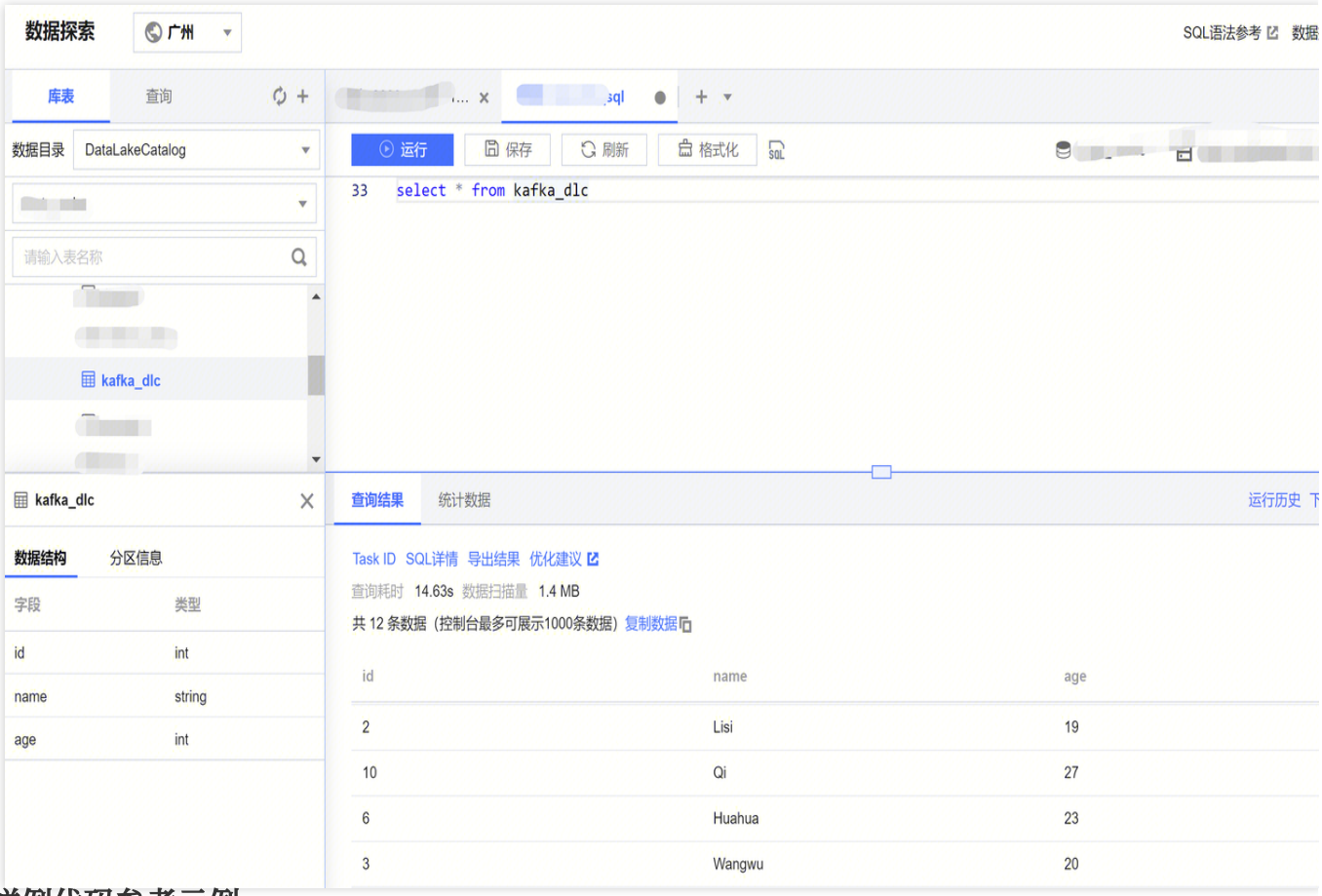
```
{"id":1,"name":"Zhangsan","age":18}
{"id":2,"name":"Lisi","age":19}
{"id":3,"name":"Wangwu","age":20}
{"id":4,"name":"Lily","age":21}
{"id":5,"name":"Lucy","age":22}
{"id":6,"name":"Huahua","age":23}
{"id":7,"name":"Wawa","age":24}
{"id":8,"name":"Mei","age":25}
{"id":9,"name":"Joi","age":26}
{"id":10,"name":"Qi","age":27}
{"id":11,"name":"Ky","age":28}
{"id":12,"name":"Mark","age":29}
```

2. 查询同步结果

打开 Flink Dashboard，单击 **Running Job > 运行Job > Checkpoint > Overview，**查看 Job 同步结果。



3. 登录 DLC 控制台，单击**数据探索**，查询目标表数据。



完整样例代码参考示例

说明：

示例中带“****”的数据请替换成开发中实际的数据。

示例1

```
<properties>
  <flink.version>1.15.4</flink.version>
  <cos.lakefs.plugin.version>1.0</cos.lakefs.plugin.version>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-java</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka</artifactId>
    <version>${flink.version}</version>
  </dependency>

  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-planner_2.12</artifactId>
```

```
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-json</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>com.qcloud.cos</groupId>
      <artifactId>lakefs-cloud-plugin</artifactId>
      <version>${cos.lakefs.plugin.version}</version>
      <exclusions>
        <exclusion>
          <groupId>com.tencentcloudapi</groupId>
          <artifactId>tencentcloud-sdk-java</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>
```

示例2

```
public class AppendIceberg {

    public static void main(String[] args) {
        // 创建执行环境 和 配置checkpoint
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnv
        env.setParallelism(1);
        env.enableCheckpointing(60000);
        env.getCheckpointConfig().setCheckpointStorage("hdfs:///flink/checkpoints")
        env.getCheckpointConfig().setCheckpointTimeout(60000);
        env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
        env.getCheckpointConfig()
                .enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpo
        EnvironmentSettings settings = EnvironmentSettings
                .newInstance()
                .inStreamingMode()
                .build();
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

        // 创建输入表
        String sourceSql = "CREATE TABLE tb_kafka_sr ( \\n"
                + "   id INT, \\n"
                + "   name STRING, \\n"
```

```
                + "   age INT \\n"
                + ") WITH ( \\n"
                + "  'connector' = 'kafka', \\n"
                + "  'topic' = 'kafka_dlc', \\n"
                + "  'properties.bootstrap.servers' = '10.0.126.***:9092', \\n" //
                + "  'properties.group.id' = 'test-group', \\n"
                + "  'scan.startup.mode' = 'earliest-offset', \\n"  // 从可能的最早偏
                + "  'format' = 'json' \\n"
                + ");";
        tEnv.executeSql(sourceSql);

        // 创建输出表
        String sinkSql = "CREATE TABLE tb_dlc_sk ( \\n"
                + "  id INT PRIMARY KEY NOT ENFORCED, \\n"
                + "  name STRING,\\n"
                + "  age INT\\n"
                + ") WITH (\\n"
                + "  'qcloud.dlc.managed.account.uid' = '1000***79117',\\n" //用户Ui
                + "  'qcloud.dlc.secret-id' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt'
                + "  'qcloud.dlc.region' = 'ap-***',\\n" // 数据库表地域信息
                + "  'qcloud.dlc.user.appid' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt
                + "  'qcloud.dlc.secret-key' = 'kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP',\
                + "  'connector' = 'iceberg-inlong', \\n"
                + "  'catalog-database' = 'test_***', \\n" // 目标数据库
                + "  'catalog-table' = 'kafka_dlc', \\n" // 目标数据表
                + "  'default-database' = 'test_***', \\n" //默认数据库
                + "  'catalog-name' = 'HYBRIS', \\n"
                + "  'catalog-impl' = 'org.apache.inlong.sort.iceberg.catalog.hybri
                + "  'uri' = 'dlc.tencentcloudapi.com', \\n"
                + "  'fs.cosn.credentials.provider' = 'org.apache.hadoop.fs.auth.Dl
                + "  'qcloud.dlc.endpoint' = 'dlc.tencentcloudapi.com', \\n"
                + "  'fs.lakefs.impl' = 'org.apache.hadoop.fs.CosFileSystem', \\n"
                + "  'fs.cosn.impl' = 'org.apache.hadoop.fs.CosFileSystem', \\n"
                + "  'fs.cosn.userinfo.region' = 'ap-guangzhou', \\n" // 使用到的COS
                + "  'fs.cosn.userinfo.secretId' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8l
                + "  'fs.cosn.userinfo.secretKey' = 'kFWYQ5WklaCYgbLtD***cyAD7sUyNi
                + "  'service.endpoint' = 'dlc.tencentcloudapi.com', \\n"
                + "  'service.secret.id' = 'AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt', \
                + "  'service.secret.key' = 'kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP', \\n
                + "  'service.region' = 'ap-***', \\n"  // 数据库表地域信息
                + "  'user.appid' = '1305424723', \\n"
                + "  'request.identity.token' = '1000***79117', \\n"
                + "  'qcloud.dlc.jdbc.url'='jdbc:dlc:dlc.internal.tencentcloudapi.c
                + ");";
        tEnv.executeSql(sinkSql);
        // 执行计算并输出
        String sql = "insert into tb_dlc_sk select * from tb_kafka_sr";
```

```
        tEnv.executeSql(sql);
    }

}
```

示例3

```
  <properties>
    <flink.version>1.15.4</flink.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>fastjson</artifactId>
      <version>2.0.22</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-java</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-clients</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-java</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
```

```
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-table-planner_2.12</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-json</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.inlong</groupId>
      <artifactId>sort-connector-iceberg-dlc</artifactId>
      <version>1.6.0</version>
      <scope>system</scope>
      <systemPath>${project.basedir}/lib/sort-connector-iceberg-dlc-1.6.0.jar</syst
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>${kafka-version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.25</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.25</version>
    </dependency>
  </dependencies>
```

示例4

```
public class KafkaToDLC {

    public static void main(String[] args) throws Exception {
        final MultipleParameterTool params = MultipleParameterTool.fromArgs(args);
        final Map<String, String> options = setOptions();
        //1.执行环境 StreamTableEnvironment,配置checkpoint
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnv
        env.setParallelism(1);
```

```
        env.enableCheckpointing(60000);
        env.getCheckpointConfig().setCheckpointStorage("hdfs:///data/checkpoints");
        env.getCheckpointConfig().setCheckpointTimeout(60000);
        env.getCheckpointConfig().setTolerableCheckpointFailureNumber(5);
        env.getCheckpointConfig()
                .enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpo
        env.getConfig().setGlobalJobParameters(params);

        //2.获取输入流
        KafkaToDLC dlcSink = new KafkaToDLC();
        DataStream<RowData> dataStreamSource = dlcSink.buildInputStream(env);

        //3.创建Hadoop配置、Catalog配置
        CatalogLoader catalogLoader = FlinkDynamicTableFactory.createCatalogLoader(
        TableLoader tableLoader = TableLoader.fromCatalog(catalogLoader,
                TableIdentifier.of(params.get(CATALOG_DATABASE.key()), params.get(C
        tableLoader.open();
        Table table = tableLoader.loadTable();
        ActionsProvider actionsProvider = FlinkDynamicTableFactory.createActionLoad
                Thread.currentThread().getContextClassLoader(), options);
        //4.创建Schema
        Schema schema = Schema.newBuilder()
                .column("id", DataTypeUtils.toInternalDataType(new IntType(false)))
                .column("name", DataTypeUtils.toInternalDataType(new VarCharType())
                .column("age", DataTypeUtils.toInternalDataType(new DateType(false)
                .primaryKey("id")
                .build();
        List<String> equalityColumns = schema.getPrimaryKey().get().getColumnNames(
        //5.配置Slink
        FlinkSink.forRowData(dataStreamSource)
                //这个 .table 也可以不写，指定tableLoader 对应的路径就可以。
                .table(table)
                .tableLoader(tableLoader)
                .equalityFieldColumns(equalityColumns)
                .metric(params.get(INLONG_METRIC.key()), params.get(INLONG_AUDIT.ke
                .action(actionsProvider)
                .tableOptions(Configuration.fromMap(options))
                //默认为false,追加数据。如果设置为true 就是覆盖数据
                .overwrite(false)
                .append();
        //6.执行同步
        env.execute("DataStream Api Write Data To Iceberg");
    }

    private static Map<String, String> setOptions() {
        Map<String, String> options = new HashMap<>();
        options.put("qcloud.dlc.managed.account.uid", "1000***79117"); //用户Uid
```

```
        options.put("qcloud.dlc.secret-id", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt");
        options.put("qcloud.dlc.region", "ap-***"); // 数据库表地域信息
        options.put("qcloud.dlc.user.appid", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt")
        options.put("qcloud.dlc.secret-key", "kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP"); /
        options.put("connector", "iceberg-inlong");
        options.put("catalog-database", "test_***"); // 目标数据库
        options.put("catalog-table", "kafka_dlc"); // 目标数据表
        options.put("default-database", "test_***"); //默认数据库
        options.put("catalog-name", "HYBRIS");
        options.put("catalog-impl", "org.apache.inlong.sort.iceberg.catalog.hybris.
        options.put("uri", "dlc.tencentcloudapi.com");
        options.put("fs.cosn.credentials.provider", "org.apache.hadoop.fs.auth.DlcC
        options.put("qcloud.dlc.endpoint", "dlc.tencentcloudapi.com");
        options.put("fs.lakefs.impl", "org.apache.hadoop.fs.CosFileSystem");
        options.put("fs.cosn.impl", "org.apache.hadoop.fs.CosFileSystem");
        options.put("fs.cosn.userinfo.region", "ap-guangzhou"); // 使用到的COS的地域信
        options.put("fs.cosn.userinfo.secretId", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJ
        options.put("fs.cosn.userinfo.secretKey", "kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP
        options.put("service.endpoint", "dlc.tencentcloudapi.com");
        options.put("service.secret.id", "AKIDwjQvBHCsKYXL3***pMdkeMsBH8lAJEt"); //
        options.put("service.secret.key", "kFWYQ5WklaCYgbLtD***cyAD7sUyNiVP"); // 用
        options.put("service.region", "ap-***");  // 数据库表地域信息
        options.put("user.appid", "1305***23");
        options.put("request.identity.token", "1000***79117");
        options.put("qcloud.dlc.jdbc.url",
                "jdbc:dlc:dlc.internal.tencentcloudapi.com?task_type,SparkSQLTask&d
        return options;
    }

    /**
     * 创建输入流
     *
     * @param env
     * @return
     */
    private DataStream<RowData> buildInputStream(StreamExecutionEnvironment env) {
        //1.配置执行环境
        EnvironmentSettings settings = EnvironmentSettings
                .newInstance()
                .inStreamingMode()
                .build();
        StreamTableEnvironment sTableEnv = StreamTableEnvironment.create(env, setti
        org.apache.flink.table.api.Table table = null;
        //2.执行SQL，获取数据输入流
        try {
            sTableEnv.executeSql(createTableSql()).print();
            table = sTableEnv.sqlQuery(transformSql());
```

```
            DataStream<Row> rowStream = sTableEnv.toChangelogStream(table);
            DataStream<RowData> rowDataDataStream = rowStream.map(new MapFunction<R
                @Override
                public RowData map(Row rows) throws Exception {
                    GenericRowData rowData = new GenericRowData(3);
                    rowData.setField(0, rows.getField(0));
                    rowData.setField(1, (String) rows.getField(1));
                    rowData.setField(2, rows.getField(2));
                    return rowData;
                }
            });
            return rowDataDataStream;
        } catch (Exception e) {
            throw new RuntimeException("kafka to dlc transform sql execute error.",
        }
    }

    private String createTableSql() {
        String tableSql = "CREATE TABLE tb_kafka_sr ( \\n"
                + "   id INT, \\n"
                + "   name STRING, \\n"
                + "   age INT \\n"
                + ") WITH ( \\n"
                + "  'connector' = 'kafka', \\n"
                + "  'topic' = 'kafka_dlc', \\n"
                + "  'properties.bootstrap.servers' = '10.0.126.30:9092', \\n"
                + "  'properties.group.id' = 'test-group-10001', \\n"
                + "  'scan.startup.mode' = 'earliest-offset', \\n"
                + "  'format' = 'json' \\n"
                + ");";
        return tableSql;
    }

    private String transformSql() {
        String transformSQL = "select * from tb_kafka_sr";
        return transformSQL;
    }
}
```

DLC 原生表常见 FAQ

最近更新时间：2024-07-31 17:35:21

为什么 Upsert 写入的 DLC 原生表（Iceberg）一定要开启数据优化？

1. DLC 原生表（Iceberg）采用的 MOR（Merge On Read）表，上游 Upsert 写入时，针对 update 的数据，会先写 delete file 标记某记录已经被删除，然后再写 data file 新增改记录。
2. 如果不提交进行合并，作业引擎在读取数据时，需要将读取原来的数据，该记录的 delete file 和新增的 data file，将三者进行合并得到最新的数据，这会导致作业需要大量的资源和时间来进行。数据优化中的小文件合并是提前将上述的文件读取回来合并，并写成新的 data file，使得作业引擎不用再进行数据文件合并而直接读取最新的文件。
3. DLC 元数据（Iceberg）采用了的快照机制，写入流程中即便是产生了新的快照也不会将历史快照清理，这依赖于数据优化的快照过期能力，将产生时间较久远的快照过期移除，从而达到释放存储空间效果，避免无用的历史数据占用存储空间。

数据优化任务出现执行超时的任务怎么处理？

系统针对数据优化任务默认设置运行超时时间（默认2小时），避免某个任务长时间占用资源而导致其他任务无法执行，当超时时间到期时该优化任务会被系统取消，根据任务类型的不同，可参考如下流程处理。
1. 如果是小文件合并任务超时，当出现连续多次超时识别的，则是数据存在了堆积，当前资源已经无法满足该表的合并，则可以临时扩展资源（或者设置该表只有优化资源为独立的资源），将历史堆积数据处理完毕后再设置回来。
2. 如果是小文件合并任务，偶尔出现任务执行超时，则是治理资源有些不足，可以适当地对数据资源扩容，并持续观察后续多个周期的治理任务是否还存在超时。当某些表偶尔出现小文件合并超时，短期内并不会对查询性能造成影响，但是不处理可能会发展为连续超时失败，到达该阶段后将影响查询性能。 DLC 默认针对小文件合并开启了分段提交，当执行超时，已经完成的部分任务仍然有机会提交成功，提交成功的合并仍然有效。
3. 如果是快照过期执行超时，快照过期执行分为两个阶段，第一个阶段从元数据中移除快照，该过程执行快照，通常不会在该阶段超时，第二个阶段将被移除快照的数据文件从存储上删除，该阶段需要逐一比较删除文件，当待删除的文件较多时，可能会出现超时。该类型的任务超时可以忽略，任务超时被移除的文件在系统会被当做孤立文件而被后续的移除孤立文件清理。
4. 如果是孤立文件执行超时，孤立文件与移除孤立文件类似，只要扫描到被删除的文件仍然是有效的，由于移除孤立文件是周期性地扫描执行，当本次任务超时后，后续的周期会继续扫描执行。

为什么 Iceberg 在 insert 写完数据后会偶尔在极短的时间内读到旧的快照？

1. Iceberg 提供了默认缓存 Catalog 的能力，默认30秒，极端情况下如果两次查询相同表间隔特别短且不在同一个 session 中执行时，在缓存没有过期和获取更新之前，有极低的概率将会查询到上一个旧快照。
2. 该参数 Iceberg 社区建议开启，DLC 在早期版本也是默认开启，该参数是为了加速任务执行，减少查询过程中对元数据的访问。 但是在极端情况下，如果两个任务读写间隔特别近，可能会出出现上述描述的情况。
3. DLC 在新版本的引擎中已经默认关闭，在结合用户场景，用户在2024年1月份之前购买的引擎如果用户需要保证数据查询到强一致，可通过如下手动关闭该参数，配置方法参考，修改引擎参数：

```
"spark.sql.catalog.DataLakeCatalog.cache-enabled": "false"
"spark.sql.catalog.DataLakeCatalog.cache.expiration-interval-ms": "0"
```

为什么建议 DLC 元原生表（Iceberg）需要进行分区？

1. 数据优化首先按照分区进行job划分，如果原生表（Iceberg）没有分区，大多数情况会改表的小文件合并都只有一个 job 执行，无法并行合并，很大程度降低小文件合并效率。
2. 如果表上游无分区字段，如何分区呢？此时可考虑 Iceberg 的 bucket 分桶，详细描述请参见 DLC 原生表核心能力。

DLC 原生表（Iceberg）写冲突如何处理？

1. Iceberg 为保证 ACID，在 commit 时要检查当前的视图是否有变化，如果有变化则判断为发生了冲突，之后回退到 commit 操作，合并当前视图，然后重新提交。
2. 系统提供了默认的冲突重试次数和时间，当发生多次 commit 操作还是发生了冲突，则将写入失败。默认冲突参数请参见 DLC 原生表核心能力。
3. 当冲突发生了，用户可以调整重试次数和重试时间。如下示例将冲突重试次数调整为10次，更多的参数含义请参见 DLC 原生表核心能力。

```
// 修改冲突重试次数为10

ALTER TABLE `DataLakeCatalog`.`axitest`.`upsert_case` SET TBLPROPERTIES('commit.ret
```

DLC 原生表（Iceberg）已经删除了，为什么存储空间容量还没有释放？

DLC 原生表（Iceberg）drop table 时元数据立即删除，数据是异步删除，先是将数据移动到回收站目录，延迟一天后数据才会从存储上移除。