尚硅谷大数据技术之数据湖Iceberg

（Iceberg & 尚硅谷 联合出品）

版本：V1.0

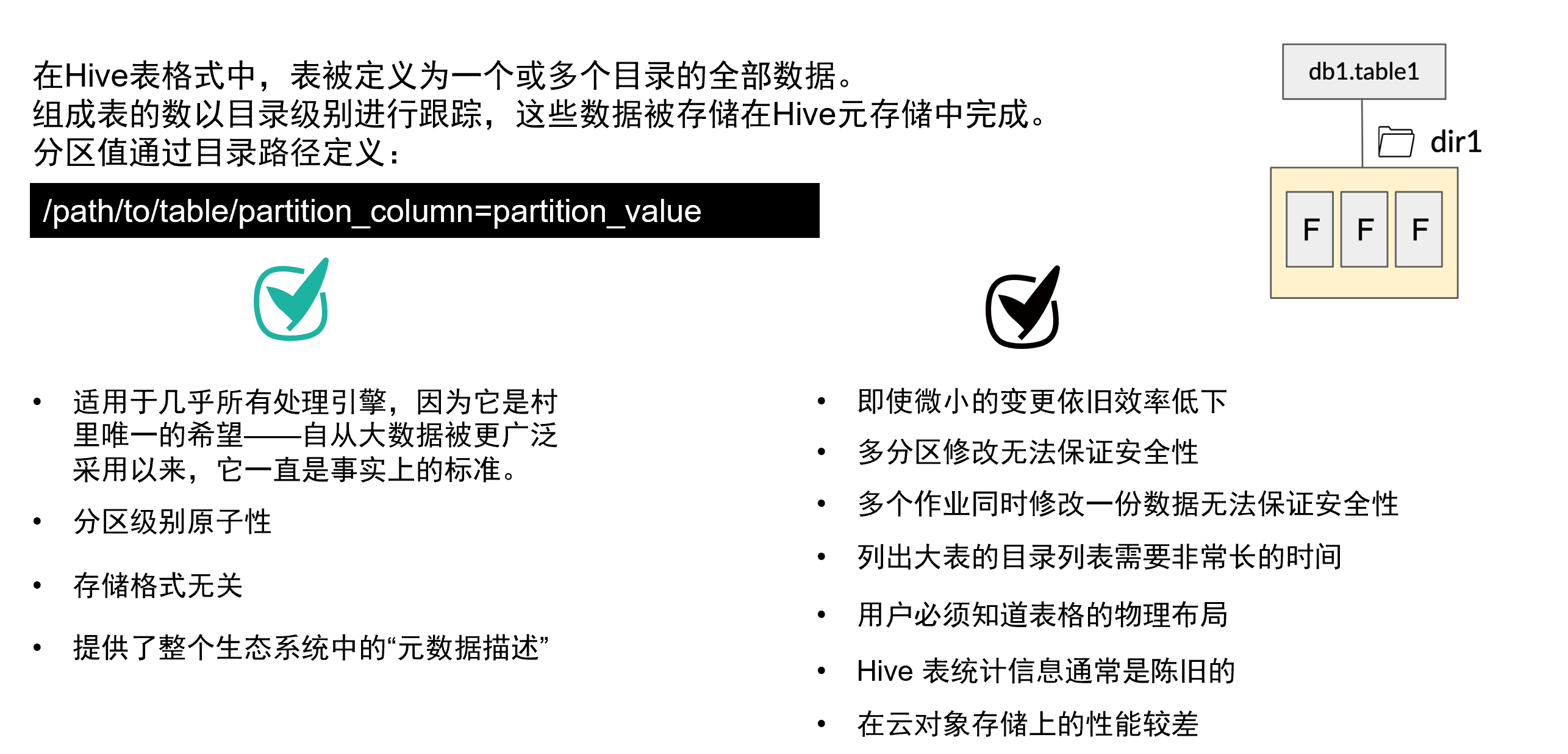
# 第1章 Iceberg简介

## 1.1 概述

为了解决数据湖中对数据管理的挑战，包括数据分区、版本控制、数据访问控制和数据模式演化等，Netflix开发了Iceberg，2018年11月16日进入Apache孵化器，2020 年5月19日从孵化器毕业，成为Apache的顶级项目。

### 为什么需要 Iceberg

**1）Hive面临的主要问题**



**2）改进要达成的目标**

图形用户界面, 文本

中度可信度描述已自动生成

### 1.1.2 Iceberg 简介

Iceberg是一个面向海量数据分析场景的开放**表格式（Table Format）**。表格式（Table Format）可以理解为元数据以及数据文件的一种组织方式，处于计算框架（Flink，Spark...）之下，数据文件之上。

文本, 信件

描述已自动生成

## 1.2 特性

### 1.2.1 数据存储、计算引擎插件化

Iceberg提供一个开放通用的表格式（Table Format）实现方案，不和特定的数据存储、计算引擎绑定。目前大数据领域的常见数据存储（HDFS、S3...），计算引擎（Flink、Spark...）都可以接入Iceberg。

在生产环境中，可选择不同的组件搭使用，甚至可以不通过计算引擎，直接读取存在文件系统上的数据。

### 1.2.2 实时流批一体

Iceberg上游组件将数据写入完成后，下游组件及时可读，可查询。可以满足实时场景.并且Iceberg同时提供了流/批读接口、流/批写接口。可以在同一个流程里, 同时处理流数据和批数据，大大简化了ETL链路。

### 1.2.3 数据表演化（Table Evolution）

Iceberg可以通过SQL或Java API的方式进行表级别模式演进。进行这些操作的时候，代价极低。 不存在读出数据重新写入或者迁移数据这种费时费力的操作。

比如在常用的Hive中，如果我们需要把一个按天分区的表，改成按小时分区。此时，不能再原表之上直接修改，只能新建一个按小时分区的表，然后再把数据Insert到新的小时分区表。而且，即使我们通过Rename的命令把新表的名字改为原表，使用原表的上次层应用, 也可能由于分区字段修改，导致需要修改 SQL，这样花费的经历是非常繁琐的。

### 1.2.4 模式演化（Schema Evolution）

Iceberg支持下面几种模式演化：

* ADD：向表或者嵌套结构增加新列
* Drop：从表中或者嵌套结构中移除一列
* Rename：重命名表中或者嵌套结构中的一列
* Update：将复杂结构(struct, map<key, value>, list)中的基本类型扩展类型长度, 比如tinyint修改成int.
* Reorder：改变列或者嵌套结构中字段的排列顺序

Iceberg保证模式演化（Schema Evolution）是没有副作用的独立操作流程, 一个元数据操作, 不会涉及到重写数据文件的过程。具体的如下:

* 增加列时候，不会从另外一个列中读取已存在的的数据
* 删除列或者嵌套结构中字段的时候，不会改变任何其他列的值
* 更新列或者嵌套结构中字段的时候，不会改变任何其他列的值
* 改变列列或者嵌套结构中字段顺序的时候，不会改变相关联的值

在表中Iceberg 使用唯一ID来定位每一列的信息。新增一个列的时候,会新分配给它一个唯一ID, 并且绝对不会使用已经被使用的ID。

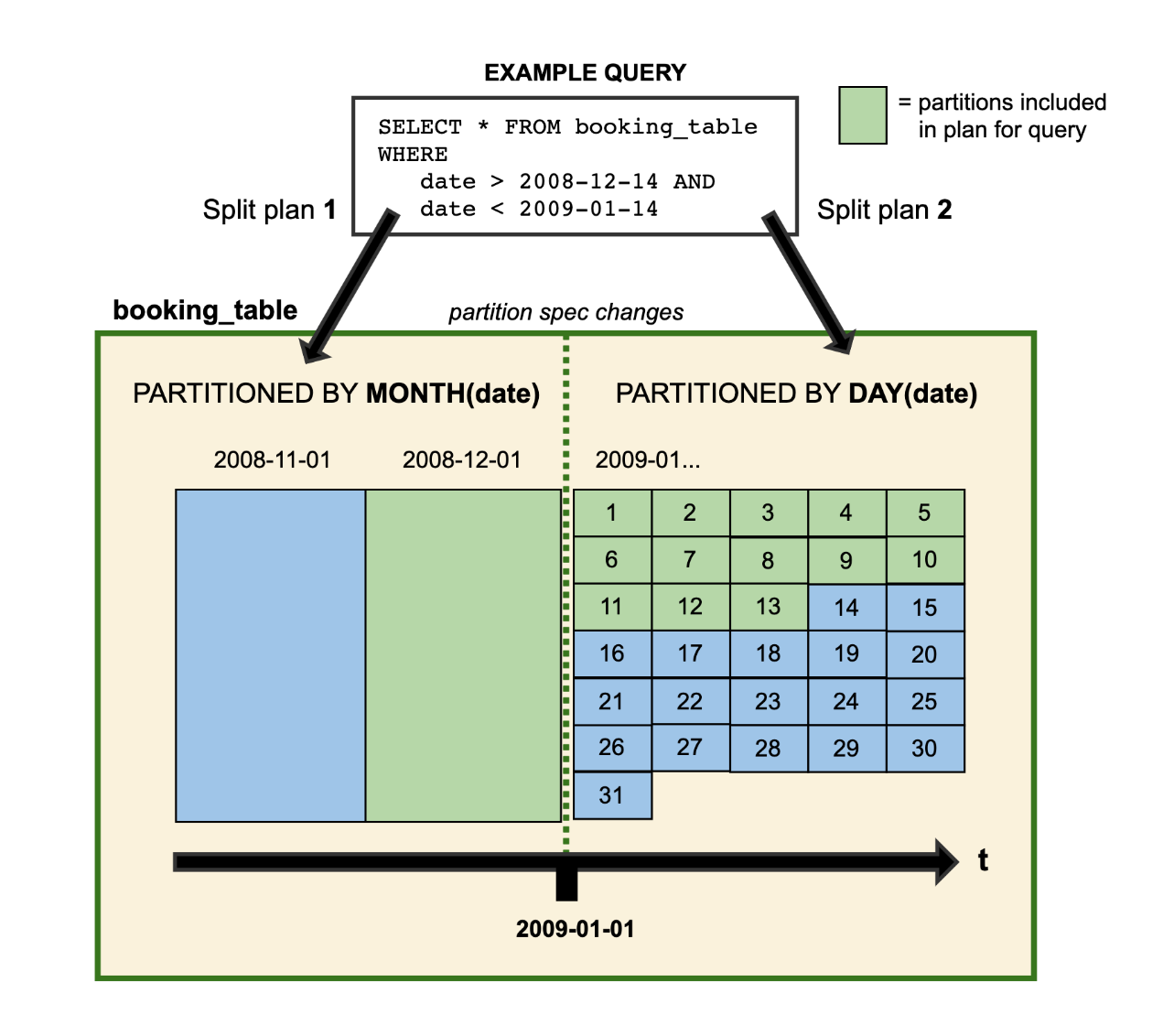
如果使用名称或者位置信息来定位列的, 都会存在一些问题, 比如使用名称的话,名称可能会重复, 使用位置的话, 不能修改顺序并且废弃的字段也不能删除。

### 1.2.5 分区演化（Partition Evolution）

Iceberg可以在一个已存在的表上直接修改，因为Iceberg的查询流程并不和分区信息直接关联。

当我们改变一个表的分区策略时，对应修改分区之前的数据不会改变, 依然会采用老的分区策略，新的数据会采用新的分区策略，也就是说同一个表会有两种分区策略，旧数据采用旧分区策略，新数据采用新新分区策略, 在元数据里两个分区策略相互独立，不重合。

在查询数据的时候，如果存在跨分区策略的情况，则会解析成两个不同执行计划，如Iceberg官网提供图所示：



图中booking\_table表2008年按月分区，进入2009年后改为按天分区，这两中分区策略共存于该表中。

借助Iceberg的隐藏分区（Hidden Partition），在写SQL 查询的时候，不需要在SQL中特别指定分区过滤条件，Iceberg会自动分区，过滤掉不需要的数据。

Iceberg分区演化操作同样是一个元数据操作, 不会重写数据文件。

### 1.2.6 列顺序演化（Sort Order Evolution）

Iceberg可以在一个已经存在的表上修改排序策略。修改了排序策略之后, 旧数据依旧采用老排序策略不变。往Iceberg里写数据的计算引擎总是会选择最新的排序策略, 但是当排序的代价极其高昂的时候, 就不进行排序了。

### 1.2.7 隐藏分区（Hidden Partition）

Iceberg的分区信息并不需要人工维护, 同时也不需要暴露给上层业务，它可以被隐藏起来. 不同于其他组件类似Hive 的分区策略, Iceberg的分区字段/策略（通过某一个字段计算出来），可以不是表的字段和表数据存储目录也没有关系。在建表或者修改分区策略之后，新的数据会自动计算所属于的分区。在查询的时候同样不用关心表的分区是什么字段/策略，只需要关注业务逻辑，Iceberg会自动过滤不需要的分区数据。

正是由于Iceberg的分区信息和表数据存储目录是独立的，使得Iceberg的表分区可以被修改,而且不会涉及到数据迁移。

### 1.2.8 镜像数据查询（Time Travel）

Iceberg提供了查询表历史某一时间点数据镜像（snapshot）的能力。可以查询出表在指定镜像\快照的表模式及数据。通过该特性可以将最新的SQL逻辑，应用到历史数据上。

### 1.2.9 支持事务（ACID）

Iceberg提供事务（ACID）保障，使其具备了upsert的能力并且使得边写边读成为可能，从而数据可以更快的被下游组件消费。通过事务保证了下游组件只能消费已commit的数据，而不会读到部分甚至未提交的数据。

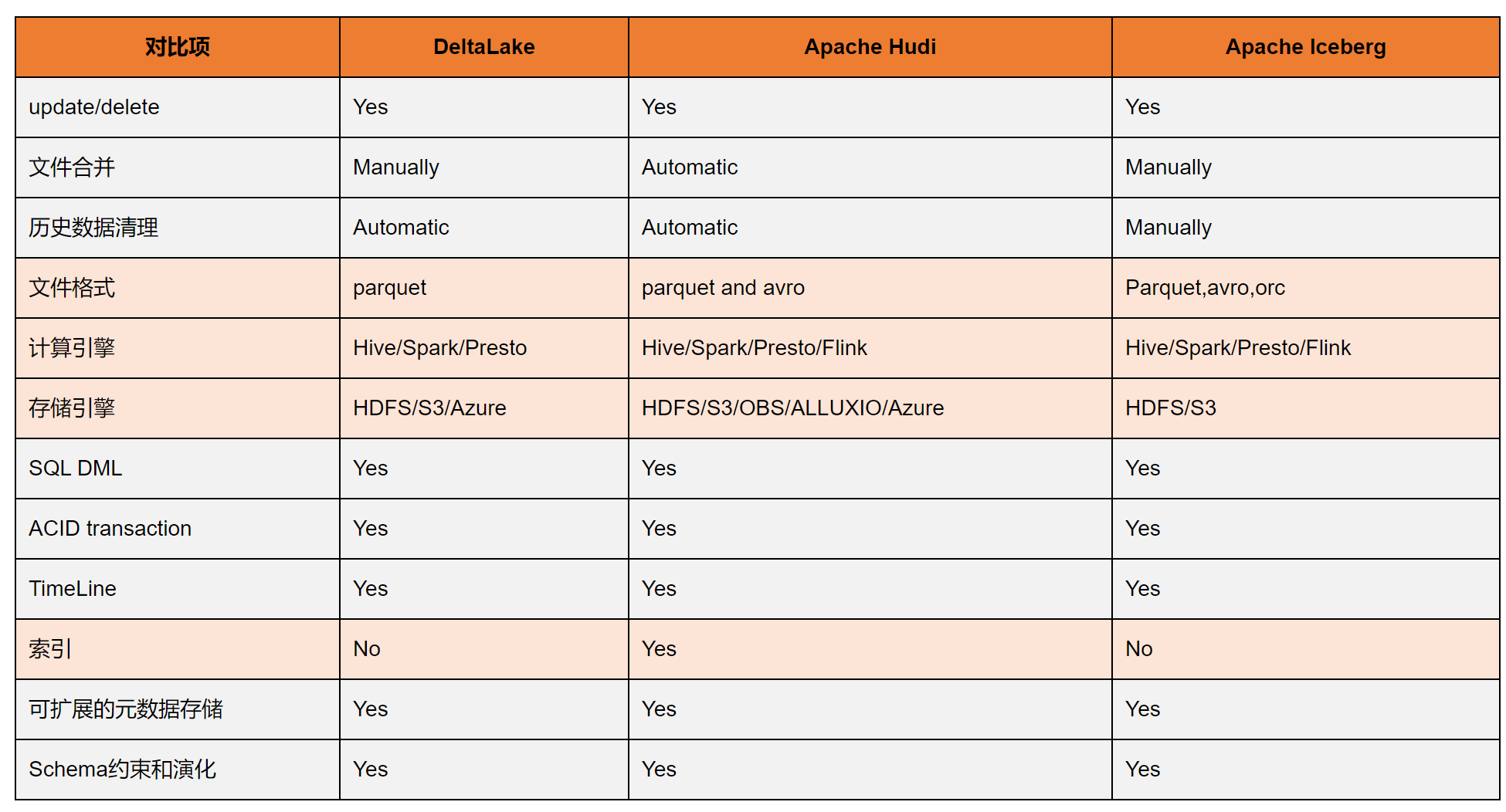
### 1.2.10 基于乐观锁的并发支持

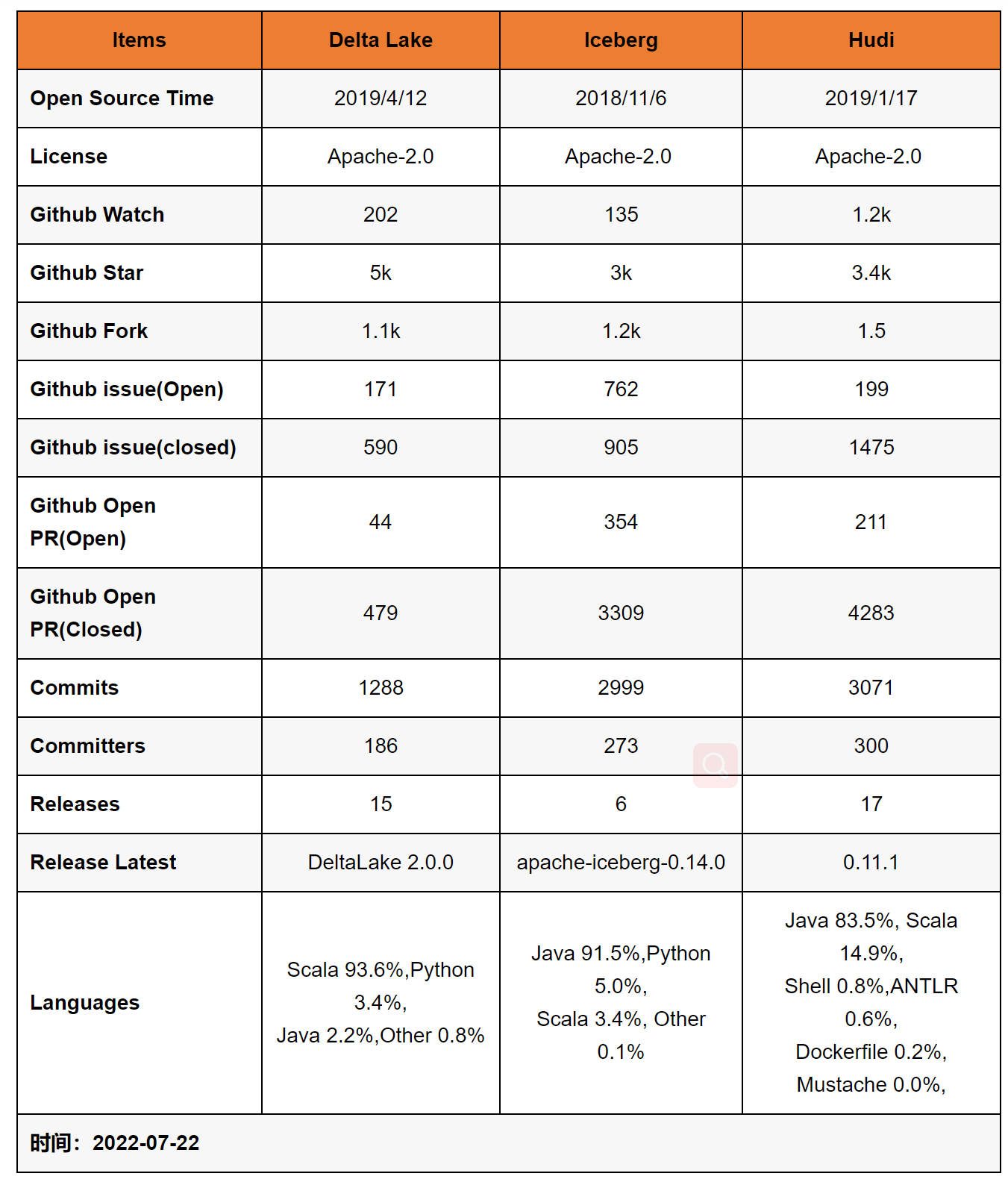
Iceberg基于乐观锁提供了多个程序并发写入的能力并且保证数据线性一致。

### 1.2.11 文件级数据剪裁

Iceberg的元数据里面提供了每个数据文件的一些统计信息，比如最大值，最小值，Count计数等等。因此，查询SQL的过滤条件除了常规的分区，列过滤，甚至可以下推到文件级别，大大加快了查询效率。

## 1.3 其他数据湖框架的对比



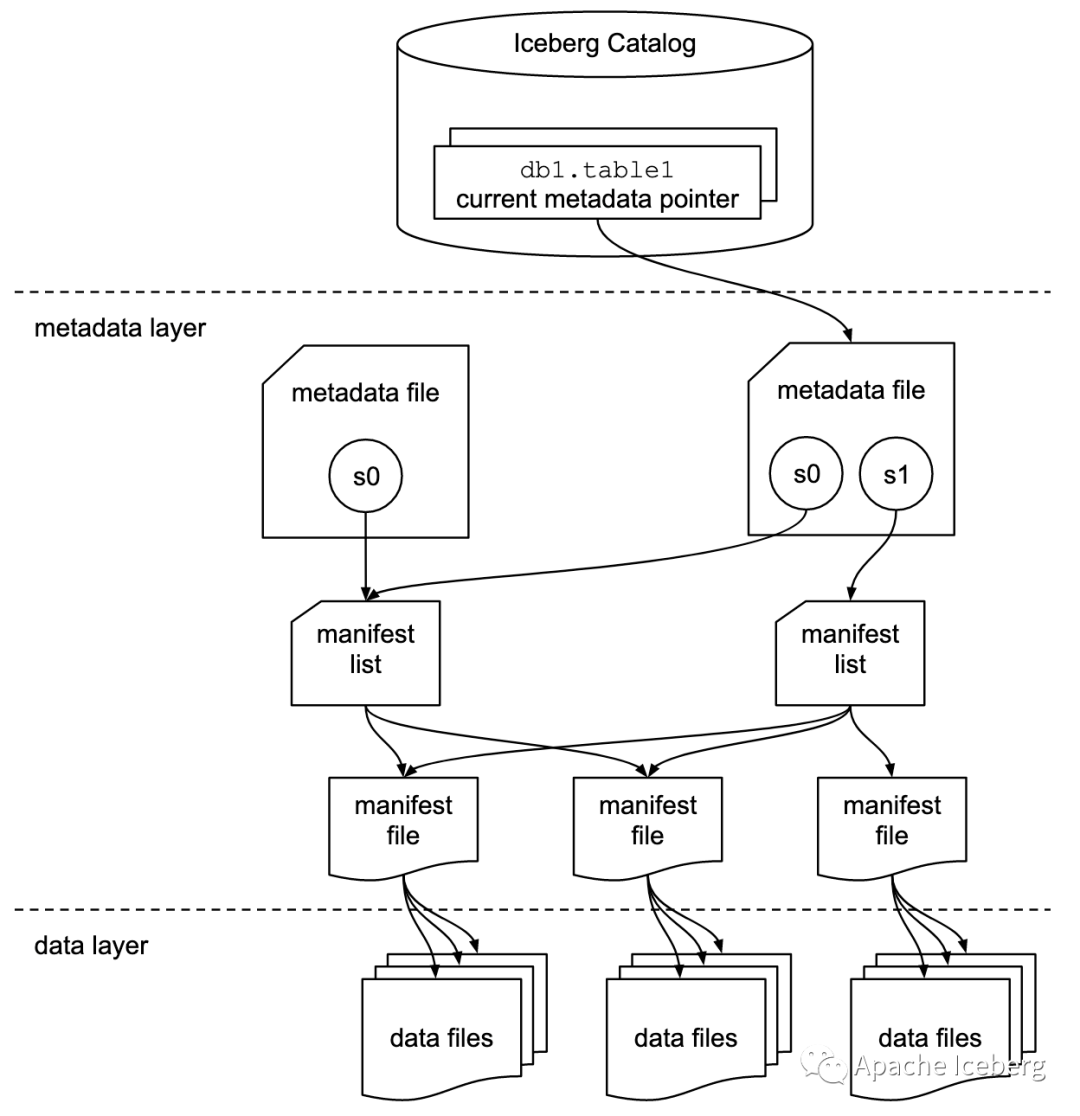


字节跳动分享的生态对比：

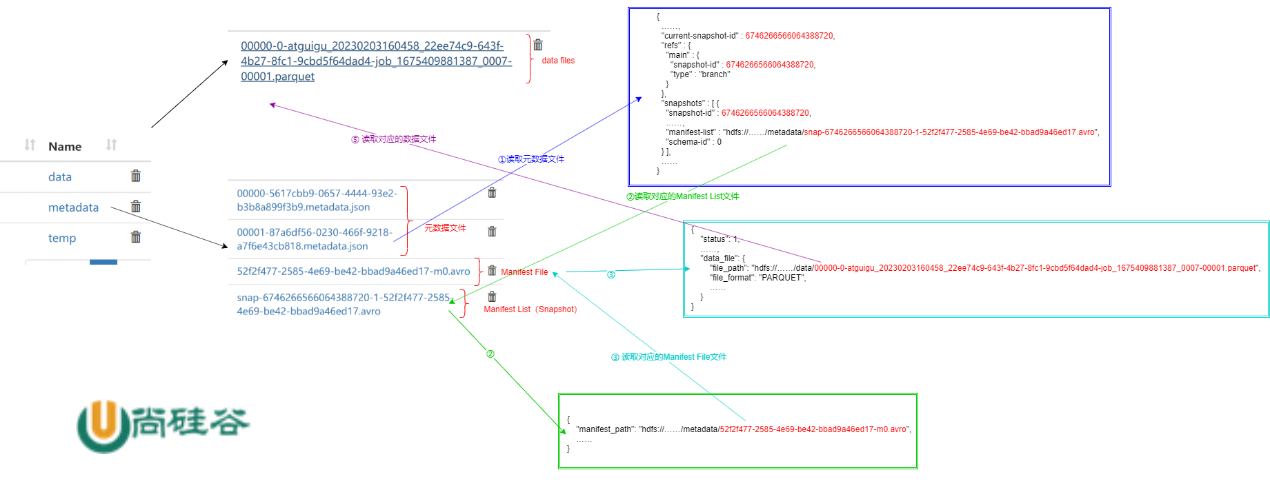




# 第2章 存储结构



动图演示：



## 2.1 数据文件 data files

数据文件是Apache Iceberg表真实存储数据的文件，一般是在表的数据存储目录的data目录下，如果我们的文件格式选择的是parquet,那么文件是以“.parquet”结尾。

例如：00000-0-atguigu\_20230203160458\_22ee74c9-643f-4b27-8fc1-9cbd5f64dad4-job\_1675409881387\_0007-00001.parquet 就是一个数据文件。

Iceberg每次更新会产生多个数据文件（data files）。

## 2.2 表快照 Snapshot

快照代表一张表在某个时刻的状态。每个快照里面会列出表在某个时刻的所有 data files 列表。data files是存储在不同的manifest files里面，manifest files是存储在一个Manifest list文件里面，而一个Manifest list文件代表一个快照。

## 2.3 清单列表 Manifest list

manifest list是一个元数据文件，它列出构建表快照（Snapshot）的清单（Manifest file）。这个元数据文件中存储的是Manifest file列表，每个Manifest file占据一行。每行中存储了Manifest file的路径、其存储的数据文件（data files）的分区范围，增加了几个数文件、删除了几个数据文件等信息，这些信息可以用来在查询时提供过滤，加快速度。

例如：snap-6746266566064388720-1-52f2f477-2585-4e69-be42-bbad9a46ed17.avro就是一个Manifest List文件。内容示例如下：

{"manifest\_path":"file:/tmp/hive8192682527333192197/table/metadata/84a46d4f-eef1-42fe-b397-78a751de5e19-m0.avro","manifest\_length":5809,"partition\_spec\_id":0,"added\_snapshot\_id":{"long":3022117681372771980},"added\_data\_files\_count":{"int":2},"existing\_data\_files\_count":{"int":0},"deleted\_data\_files\_count":{"int":0},"partitions":{"array":[]},"added\_rows\_count":{"long":3},"existing\_rows\_count":{"long":0},"deleted\_rows\_count":{"long":0}}

{"manifest\_path":"file:/tmp/hive8192682527333192197/table/metadata/942151bb-a154-48c1-a335-9a0d1677e56d-m0.avro","manifest\_length":5812,"partition\_spec\_id":0,"added\_snapshot\_id":{"long":2632728956091659683},"added\_data\_files\_count":{"int":2},"existing\_data\_files\_count":{"int":0},"deleted\_data\_files\_count":{"int":0},"partitions":{"array":[]},"added\_rows\_count":{"long":3},"existing\_rows\_count":{"long":0},"deleted\_rows\_count":{"long":0}}

## 2.4 清单文件 Manifest file

Manifest file也是一个元数据文件，它列出组成快照（snapshot）的数据文件（data files）的列表信息。每行都是每个数据文件的详细描述，包括数据文件的状态、文件路径、分区信息、列级别的统计信息（比如每列的最大最小值、空值数等）、文件的大小以及文件里面数据行数等信息。其中列级别的统计信息可以在扫描表数据时过滤掉不必要的文件。

Manifest file是以avro格式进行存储的，以“.avro”后缀结尾，例如：52f2f477-2585-4e69-be42-bbad9a46ed17-m0.avro。

内容示例如下：

{"status":1,"snapshot\_id":{"long":2632728956091659683},"data\_file":{"file\_path":"file:/tmp/hive8192682527333192197/table/data/00000-0-29e3896b-0813-48e0-a6c2-93ec4d39a652-00001.orc","file\_format":"ORC","partition":{},"record\_count":1,"file\_size\_in\_bytes":352,"block\_size\_in\_bytes":67108864,"column\_sizes":{"array":[{"key":1,"value":6},{"key":2,"value":11}]},"value\_counts":{"array":[{"key":1,"value":1},{"key":2,"value":1}]},"null\_value\_counts":{"array":[{"key":1,"value":0},{"key":2,"value":0}]},"nan\_value\_counts":{"array":[]},"lower\_bounds":{"array":[{"key":1,"value":"\u0001\u0000\u0000\u0000"},{"key":2,"value":"hr"}]},"upper\_bounds":{"array":[{"key":1,"value":"\u0001\u0000\u0000\u0000"},{"key":2,"value":"hr"}]},"key\_metadata":null,"split\_offsets":{"array":[3]},"sort\_order\_id":{"int":0}}}

{"status":1,"snapshot\_id":{"long":2632728956091659683},"data\_file":{"file\_path":"file:/tmp/hive8192682527333192197/table/data/00001-1-29e3896b-0813-48e0-a6c2-93ec4d39a652-00001.orc","file\_format":"ORC","partition":{},"record\_count":2,"file\_size\_in\_bytes":390,"block\_size\_in\_bytes":67108864,"column\_sizes":{"array":[{"key":1,"value":11},{"key":2,"value":19}]},"value\_counts":{"array":[{"key":1,"value":2},{"key":2,"value":2}]},"null\_value\_counts":{"array":[{"key":1,"value":1},{"key":2,"value":0}]},"nan\_value\_counts":{"array":[]},"lower\_bounds":{"array":[{"key":1,"value":"\u0002\u0000\u0000\u0000"},{"key":2,"value":"hardware"}]},"upper\_bounds":{"array":[{"key":1,"value":"\u0002\u0000\u0000\u0000"},{"key":2,"value":"hr"}]},"key\_metadata":null,"split\_offsets":{"array":[3]},"sort\_order\_id":{"int":0}}}

# 第3章 与 Hive集成

## 3.1 环境准备

**1）Hive与Iceberg的版本对应关系如下**

|  |  |  |
| --- | --- | --- |
| Hive 版本 | 官方推荐Hive版本 | Iceberg 版本 |
| 2.x | 2.3.8 | 0.8.0-incubating – 1.1.0 |
| 3.x | 3.1.2 | 0.10.0 – 1.1.0 |

Iceberg与Hive 2和Hive 3.1.2/3的集成，支持以下特性：

* 创建表
* 删除表
* 读取表
* 插入表（INSERT into）

更多功能需要Hive 4.x（目前alpha版本）才能支持。

**2）上传jar包，拷贝到Hive的auxlib目录中**

mkdir auxlib

cp iceberg-hive-runtime-1.1.0.jar /opt/module/hive/auxlib

cp libfb303-0.9.3.jar /opt/module/hive/auxlib

**3）修改hive-site.xml，添加配置项**

<property>

<name>iceberg.engine.hive.enabled</name>

<value>true</value>

</property>

<property>

<name>hive.aux.jars.path</name>

<value>/opt/module/hive/auxlib</value>

</property>

使用TEZ引擎注意事项：

（1）使用Hive版本>=3.1.2，需要TEZ版本>=0.10.1

（2）指定tez更新配置：

<property>

<name>tez.mrreader.config.update.properties</name>

<value>hive.io.file.readcolumn.names,hive.io.file.readcolumn.ids</value>

</property>

（3）从Iceberg 0.11.0开始，如果Hive使用Tez引擎，需要关闭向量化执行：

<property>

<name>hive.vectorized.execution.enabled</name>

<value>false</value>

</property>

**4）启动HMS服务**

**5）启动 Hadoop**

## 3.2 创建和管理 Catalog

Iceberg支持多种不同的Catalog类型，例如:Hive、Hadoop、亚马逊的AWS Glue和自定义Catalog。

根据不同配置，分为三种情况：

* 没有设置iceberg.catalog，默认使用HiveCatalog

|  |  |
| --- | --- |
| 配置项 | 说明 |
| iceberg.catalog.<catalog\_name>.type | Catalog的类型: hive, hadoop, 如果使用自定义Catalog，则不设置 |
| iceberg.catalog.<catalog\_name>.catalog-impl | Catalog的实现类, 如果上面的type没有设置，则此参数必须设置 |
| iceberg.catalog.<catalog\_name>.<key> | Catalog的其他配置项 |

* 设置了 iceberg.catalog的类型，使用指定的Catalog类型，如下表格：
* 设置 iceberg.catalog=location\_based\_table，直接通过指定的根路径来加载Iceberg表

### 3.1.1 默认使用 HiveCatalog

CREATE TABLE iceberg\_test1 (i int)

STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler';

INSERT INTO iceberg\_test1 values(1);

查看HDFS可以发现，表目录在默认的hive仓库路径下。

### 3.1.2 指定 Catalog 类型

**1）使用 HiveCatalog**

set iceberg.catalog.iceberg\_hive.type=hive;

set iceberg.catalog.iceberg\_hive.uri=thrift://hadoop1:9083;

set iceberg.catalog.iceberg\_hive.clients=10;

set iceberg.catalog.iceberg\_hive.warehouse=hdfs://hadoop1:8020/warehouse/iceberg-hive;

CREATE TABLE iceberg\_test2 (i int)

STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'

TBLPROPERTIES('iceberg.catalog'='iceberg\_hive');

INSERT INTO iceberg\_test2 values(1);

**2）使用 HadoopCatalog**

set iceberg.catalog.iceberg\_hadoop.type=hadoop;

set iceberg.catalog.iceberg\_hadoop.warehouse=hdfs://hadoop1:8020/warehouse/iceberg-hadoop;

CREATE TABLE iceberg\_test3 (i int)

STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'

LOCATION 'hdfs://hadoop1:8020/warehouse/iceberg-hadoop/default/iceberg\_test3'

TBLPROPERTIES('iceberg.catalog'='iceberg\_hadoop');

INSERT INTO iceberg\_test3 values(1);

### 3.1.3 指定路径加载

如果HDFS中已经存在iceberg格式表，我们可以通过在Hive中创建Icerberg格式表指定对应的location路径映射数据。

CREATE EXTERNAL TABLE iceberg\_test4 (i int)

STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'

LOCATION 'hdfs://hadoop1:8020/warehouse/iceberg-hadoop/default/iceberg\_test3'

TBLPROPERTIES ('iceberg.catalog'='location\_based\_table');

## 3.3 基本操作

### 3.3.1 创建表

**1）创建外部表**

CREATE EXTERNAL TABLE iceberg\_create1 (i int)

STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler';

describe formatted iceberg\_create1;

**2）创建内部表**

CREATE TABLE iceberg\_create2 (i int)

STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler';

describe formatted iceberg\_create2;

**3）创建分区表**

CREATE EXTERNAL TABLE iceberg\_create3 (id int,name string)

PARTITIONED BY (age int)

STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler';

describe formatted iceberg\_create3;

Hive语法创建分区表，不会在HMS中创建分区，而是将分区数据转换为Iceberg标识分区。这种情况下不能使用Iceberg的分区转换，例如：days(timestamp)，如果想要使用Iceberg格式表的分区转换标识分区，需要使用Spark或者Flink引擎创建表。

### 3.3.2 修改表

只支持HiveCatalog表修改表属性，Iceberg表属性和Hive表属性存储在HMS中是同步的。

ALTER TABLE iceberg\_create1 SET TBLPROPERTIES('external.table.purge'='FALSE');

### 3.3.3 插入表

支持标准单表INSERT INTO操作

INSERT INTO iceberg\_create2 VALUES (1);

INSERT INTO iceberg\_create1 select \* from iceberg\_create2;

在HIVE 3.x中，INSERT OVERWRITE虽然能执行，但其实是追加。

### 3.3.4 删除表

DROP TABLE iceberg\_create1;

# 第4章 与Spark SQL集成

## 4.1 环境准备

### 4.1.1 安装 Spark

**1）****Spark与Iceberg的版本对应关系如下**

|  |  |
| --- | --- |
| Spark 版本 | Iceberg 版本 |
| 2.4 | 0.7.0-incubating – 1.1.0 |
| 3.0 | 0.9.0 – 1.0.0 |
| 3.1 | 0.12.0 – 1.1.0 |
| 3.2 | 0.13.0 – 1.1.0 |
| 3.3 | 0.14.0 – 1.1.0 |

**2）上传并解压Spark安装包**

tar -zxvf spark-3.3.1-bin-hadoop3.tgz -C /opt/module/

mv /opt/module/spark-3.3.1-bin-hadoop3 /opt/module/spark-3.3.1

**3）配置环境变量**

sudo vim /etc/profile.d/my\_env.sh

export SPARK\_HOME=/opt/module/spark-3.3.1

export PATH=$PATH:$SPARK\_HOME/bin

source /etc/profile.d/my\_env.sh

**4）拷贝iceberg的jar包到Spark的jars目录**

cp /opt/software/iceberg/iceberg-spark-runtime-3.3\_2.12-1.1.0.jar /opt/module/spark-3.3.1/jars

### 4.1.2 启动 Hadoop

（略）

## 4.2 Spark 配置 Catalog

Spark中支持两种Catalog的设置：hive和hadoop，Hive Catalog就是Iceberg表使用HMS来存储表的部分元数据信息，并使用其锁定机制来完成事务管理，Hadoop Catalog需要指定Iceberg格式表存储路径。

vim spark-defaults.conf

### 4.2.1 Hive Catalog

spark.sql.catalog.hive\_prod = org.apache.iceberg.spark.SparkCatalog

spark.sql.catalog.hive\_prod.type = hive

spark.sql.catalog.hive\_prod.uri = thrift://hadoop1:9083

use hive\_prod.db;

### 4.2.2 Hadoop Catalog

spark.sql.catalog.hadoop\_prod = org.apache.iceberg.spark.SparkCatalog

spark.sql.catalog.hadoop\_prod.type = hadoop

spark.sql.catalog.hadoop\_prod.warehouse = hdfs://hadoop1:8020/warehouse/spark-iceberg

use hadoop\_prod.db;

## 4.3 SQL 操作

### 4.3.1 创建表

use hadoop\_prod;

create database default;

use default;

CREATE TABLE hadoop\_prod.default.sample1 (

id bigint COMMENT 'unique id',

data string)

USING iceberg

* PARTITIONED BY (partition-expressions) ：配置分区
* LOCATION '(fully-qualified-uri)' ：指定表路径
* COMMENT 'table documentation' ：配置表备注
* TBLPROPERTIES ('key'='value', ...) ：配置表属性

表属性：<https://iceberg.apache.org/docs/latest/configuration/>

对Iceberg表的每次更改都会生成一个新的元数据文件（json文件）以提供原子性。默认情况下，旧元数据文件作为历史文件保存不会删除。

如果要自动清除元数据文件，在表属性中设置write.metadata.delete-after-commit.enabled=true。这将保留一些元数据文件（直到write.metadata.previous-versions-max），并在每个新创建的元数据文件之后删除旧的元数据文件。

**1）创建分区表**

（1）分区表

CREATE TABLE hadoop\_prod.default.sample2 (

id bigint,

data string,

category string)

USING iceberg

PARTITIONED BY (category)

（2）创建隐藏分区表

CREATE TABLE hadoop\_prod.default.sample3 (

id bigint,

data string,

category string,

ts timestamp)

USING iceberg

PARTITIONED BY (bucket(16, id), days(ts), category)

支持的转换有:

* years(ts)：按年划分
* months(ts)：按月划分
* days(ts)或date(ts)：等效于dateint分区
* hours(ts)或date\_hour(ts)：等效于dateint和hour分区
* bucket(N, col)：按哈希值划分mod N个桶
* truncate(L, col)：按截断为L的值划分

字符串被截断为给定的长度

整型和长型截断为bin: truncate(10, i)生成分区0,10,20,30，…

**2）使用 CTAS 语法建表**

CREATE TABLE hadoop\_prod.default.sample4

USING iceberg

AS SELECT \* from hadoop\_prod.default.sample3

不指定分区就是无分区，需要重新指定分区、表属性：

CREATE TABLE hadoop\_prod.default.sample5

USING iceberg

PARTITIONED BY (bucket(8, id), hours(ts), category)

TBLPROPERTIES ('key'='value')

AS SELECT \* from hadoop\_prod.default.sample3

**3）使用 Replace table 建表**

REPLACE TABLE hadoop\_prod.default.sample5

USING iceberg

AS SELECT \* from hadoop\_prod.default.sample3

REPLACE TABLE hadoop\_prod.default.sample5

USING iceberg

PARTITIONED BY (part)

TBLPROPERTIES ('key'='value')

AS SELECT \* from hadoop\_prod.default.sample3

CREATE OR REPLACE TABLE hadoop\_prod.default.sample6

USING iceberg

AS SELECT \* from hadoop\_prod.default.sample3

### 4.3.2 删除表

对于HadoopCatalog而言，运行DROP TABLE将从catalog中删除表并删除表内容。

CREATE EXTERNAL TABLE hadoop\_prod.default.sample7 (

id bigint COMMENT 'unique id',

data string)

USING iceberg

INSERT INTO hadoop\_prod.default.sample7 values(1,'a')

DROP TABLE hadoop\_prod.default.sample7

对于HiveCatalog而言：

* 在0.14之前，运行DROP TABLE将从catalog中删除表并删除表内容。
* 从0.14开始，DROP TABLE只会从catalog中删除表，不会删除数据。为了删除表内容，应该使用DROP table PURGE。

CREATE TABLE hive\_prod.default.sample7 (

id bigint COMMENT 'unique id',

data string)

USING iceberg

INSERT INTO hive\_prod.default.sample7 values(1,'a')

**1）删除表**

DROP TABLE hive\_prod.default.sample7

**2）删除表和数据**

DROP TABLE hive\_prod.default.sample7 PURGE

### 4.3.3 修改表

Iceberg在Spark 3中完全支持ALTER TABLE，包括：

* 重命名表
* 设置或删除表属性
* 添加、删除和重命名列
* 添加、删除和重命名嵌套字段
* 重新排序顶级列和嵌套结构字段
* 扩大int、float和decimal字段的类型
* 将必选列变为可选列

此外，还可以使用SQL扩展来添加对分区演变的支持和设置表的写顺序。

CREATE TABLE hive\_prod.default.sample1 (

id bigint COMMENT 'unique id',

data string)

USING iceberg

**1）修改表名（不支持修改HadoopCatalog的表名）**

ALTER TABLE hive\_prod.default.sample1 RENAME TO hive\_prod.default.sample2

**2）修改表属性**

（1）修改表属性

ALTER TABLE hive\_prod.default.sample1 SET TBLPROPERTIES (

'read.split.target-size'='268435456'

)

ALTER TABLE hive\_prod.default.sample1 SET TBLPROPERTIES (

'comment' = 'A table comment.'

)

（2）删除表属性

ALTER TABLE hive\_prod.default.sample1 UNSET TBLPROPERTIES ('read.split.target-size')

**3）添加列**

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMNS (

category string comment 'new\_column'

)

-- 添加struct类型的列

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN point struct<x: double, y: double>;

-- 往struct类型的列中添加字段

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN point.z double

-- 创建struct的嵌套数组列

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN points array<struct<x: double, y: double>>;

-- 在数组中的结构中添加一个字段。使用关键字'element'访问数组的元素列。

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN points.element.z double

-- 创建一个包含Map类型的列，key和value都为struct类型

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN pointsm map<struct<x: int>, struct<a: int>>;

-- 在Map类型的value的struct中添加一个字段。

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN pointsm.value.b int

在Spark 2.4.4及以后版本中，可以通过添加FIRST或AFTER子句在任何位置添加列：

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN new\_column1 bigint AFTER id

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMN new\_column2 bigint FIRST

**4）修改列**

（1）修改列名

ALTER TABLE hadoop\_prod.default.sample1 RENAME COLUMN data TO data1

（2）Alter Column修改类型（只允许安全的转换）

ALTER TABLE hadoop\_prod.default.sample1

ADD COLUMNS (

idd int

)

ALTER TABLE hadoop\_prod.default.sample1 ALTER COLUMN idd TYPE bigint

（3）Alter Column 修改列的注释

ALTER TABLE hadoop\_prod.default.sample1 ALTER COLUMN id TYPE double COMMENT 'a'

ALTER TABLE hadoop\_prod.default.sample1 ALTER COLUMN id COMMENT 'b'

（4）Alter Column修改列的顺序

ALTER TABLE hadoop\_prod.default.sample1 ALTER COLUMN id FIRST

ALTER TABLE hadoop\_prod.default.sample1 ALTER COLUMN new\_column2 AFTER new\_column1

（5）Alter Column修改列是否允许为null

ALTER TABLE hadoop\_prod.default.sample1 ALTER COLUMN id DROP NOT NULL

ALTER COLUMN不用于更新struct类型。使用ADD COLUMN和DROP COLUMN添加或删除struct类型的字段。

**5）删除列**

ALTER TABLE hadoop\_prod.default.sample1 DROP COLUMN idd

ALTER TABLE hadoop\_prod.default.sample1 DROP COLUMN point.z

**6）添加分区（Spark3，需要配置扩展）**

vim spark-default.conf

spark.sql.extensions =org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions

重新进入spark-sql shell：

ALTER TABLE hadoop\_prod.default.sample1 ADD PARTITION FIELD category

ALTER TABLE hadoop\_prod.default.sample1 ADD PARTITION FIELD bucket(16, id)

ALTER TABLE hadoop\_prod.default.sample1 ADD PARTITION FIELD truncate(data, 4)

ALTER TABLE hadoop\_prod.default.sample1 ADD PARTITION FIELD years(ts)

ALTER TABLE hadoop\_prod.default.sample1 ADD PARTITION FIELD bucket(16, id) AS shard

**7）删除分区（Spark3，需要配置扩展）**

ALTER TABLE hadoop\_prod.default.sample1 DROP PARTITION FIELD category

ALTER TABLE hadoop\_prod.default.sample1 DROP PARTITION FIELD bucket(16, id)

ALTER TABLE hadoop\_prod.default.sample1 DROP PARTITION FIELD truncate(data, 4)

ALTER TABLE hadoop\_prod.default.sample1 DROP PARTITION FIELD years(ts)

ALTER TABLE hadoop\_prod.default.sample1 DROP PARTITION FIELD shard

注意，尽管删除了分区，但列仍然存在于表结构中。

删除分区字段是元数据操作，不会改变任何现有的表数据。新数据将被写入新的分区，但现有数据将保留在旧的分区布局中。

当分区发生变化时，动态分区覆盖行为也会发生变化。例如，如果按天划分分区，而改为按小时划分分区，那么覆盖将覆盖每小时划分的分区，而不再覆盖按天划分的分区。

删除分区字段时要小心，可能导致元数据查询失败或产生不同的结果。

**8）修改分区（****Spark3，需要配置扩展 spark.sql.extensions）**

ALTER TABLE hadoop\_prod.default.sample1 REPLACE PARTITION FIELD bucket(16, id) WITH bucket(8, id)

**9）修改表的写入顺序**

ALTER TABLE hadoop\_prod.default.sample1 WRITE ORDERED BY category, id

ALTER TABLE hadoop\_prod.default.sample1 WRITE ORDERED BY category ASC, id DESC

ALTER TABLE hadoop\_prod.default.sample1 WRITE ORDERED BY category ASC NULLS LAST, id DESC NULLS FIRST

表写顺序不能保证查询的数据顺序。它只影响数据写入表的方式。

WRITE ORDERED BY设置了一个全局排序，即跨任务的行排序，就像在INSERT命令中使用ORDER BY一样：

INSERT INTO hadoop\_prod.default.sample1

SELECT id, data, category, ts FROM another\_table

ORDER BY ts, category

要在每个任务内排序，而不是跨任务排序，使用local ORDERED BY：

ALTER TABLE hadoop\_prod.default.sample1 WRITE LOCALLY ORDERED BY category, id

**10）按分区并行写入**

ALTER TABLE hadoop\_prod.default.sample1 WRITE DISTRIBUTED BY PARTITION

ALTER TABLE hadoop\_prod.default.sample1 WRITE DISTRIBUTED BY PARTITION LOCALLY ORDERED BY category, id

### 4.3.4 插入数据

CREATE TABLE hadoop\_prod.default.a (

id bigint,

count bigint)

USING iceberg

CREATE TABLE hadoop\_prod.default.b (

id bigint,

count bigint,

flag string)

USING iceberg

**1）Insert Into**

INSERT INTO hadoop\_prod.default.a VALUES (1, 1), (2, 2), (3, 3);

INSERT INTO hadoop\_prod.default.b VALUES (1, 1, 'a'), (2, 2, 'b'), (4, 4, 'd');

**2）MERGE INTO行级更新****（Spark3，需要配置扩展 spark.sql.extensions）**

MERGE INTO hadoop\_prod.default.a t

USING (SELECT \* FROM hadoop\_prod.default.b) u ON t.id = u.id

WHEN MATCHED AND u.flag='b' THEN UPDATE SET t.count = t.count + u.count

WHEN MATCHED AND u.flag='a' THEN DELETE

WHEN NOT MATCHED THEN INSERT (id,count) values (u.id,u.count)

简单的更新和删除数据，详见 4.5。

### 4.3.5 查询数据

**1）普通查询**

SELECT count(1) as count, data

FROM local.db.table

GROUP BY data

**2）查询元数据**

// 查询表快照

SELECT \* FROM hadoop\_prod.default.a.snapshots

// 查询数据文件信息

SELECT \* FROM hadoop\_prod.default.a.files

// 查询表历史

SELECT \* FROM hadoop\_prod.default.a.history

// 查询 manifest

ELECT \* FROM hadoop\_prod.default.a.manifests

### 4.3.6 存储过程

Procedures可以通过CALL从任何已配置的Iceberg Catalog中使用，需要配置扩展 spark.sql.extensions。所有Procedures都在namespace中。

1. **语法**

按照参数名传参

CALL catalog\_name.system.procedure\_name(arg\_name\_2 => arg\_2, arg\_name\_1 => arg\_1)

当按位置传递参数时，如果结束参数是可选的，则只有结束参数可以省略。

CALL catalog\_name.system.procedure\_name(arg\_1, arg\_2, ... arg\_n)

**2）快照管理**

（1）回滚到指定的快照id

CALL hadoop\_prod.system.rollback\_to\_snapshot('default.a', 7601163594701794741)

（2）回滚到指定时间的快照

CALL hadoop\_prod.system.rollback\_to\_timestamp('db.sample', TIMESTAMP '2021-06-30 00:00:00.000')

（3）设置表的当前快照ID

CALL hadoop\_prod.system.set\_current\_snapshot('db.sample', 1)

（4）从快照变为当前表状态

CALL hadoop\_prod.system.cherrypick\_snapshot('default.a', 7629160535368763452)

CALL hadoop\_prod.system.cherrypick\_snapshot(snapshot\_id => 7629160535368763452, table => 'default.a' )

**3）元数据管理**

（1）删除早于指定日期和时间的快照，但保留最近100个快照

CALL hive\_prod.system.expire\_snapshots('db.sample', TIMESTAMP '2021-06-30 00:00:00.000', 100)

（2）删除Iceberg表中任何元数据文件中没有引用的文件

#列出所有需要删除的候选文件

CALL catalog\_name.system.remove\_orphan\_files(table => 'db.sample', dry\_run => true)

#删除指定目录中db.sample表不知道的任何文件

CALL catalog\_name.system.remove\_orphan\_files(table => 'db.sample', location => 'tablelocation/data')

（3）合并数据文件（合并小文件）

CALL catalog\_name.system.rewrite\_data\_files('db.sample')

CALL catalog\_name.system.rewrite\_data\_files(table => 'db.sample', strategy => 'sort', sort\_order => 'id DESC NULLS LAST,name ASC NULLS FIRST')

CALL catalog\_name.system.rewrite\_data\_files(table => 'db.sample', strategy => 'sort', sort\_order => 'zorder(c1,c2)')

CALL catalog\_name.system.rewrite\_data\_files(table => 'db.sample', options => map('min-input-files','2'))

CALL catalog\_name.system.rewrite\_data\_files(table => 'db.sample', where => 'id = 3 and name = "foo"')

（4）重写表清单来优化执行计划

CALL catalog\_name.system.rewrite\_manifests('db.sample')

#重写表db中的清单。并禁用Spark缓存的使用。这样做可以避免执行程序上的内存问题。

CALL catalog\_name.system.rewrite\_manifests('db.sample', false)

**4）迁移表**

（1）快照

CALL catalog\_name.system.snapshot('db.sample', 'db.snap')

CALL catalog\_name.system.snapshot('db.sample', 'db.snap', '/tmp/temptable/')

（2）迁移

CALL catalog\_name.system.migrate('spark\_catalog.db.sample', map('foo', 'bar'))

CALL catalog\_name.system.migrate('db.sample')

（3）添加数据文件

CALL spark\_catalog.system.add\_files(

table => 'db.tbl',

source\_table => 'db.src\_tbl',

partition\_filter => map('part\_col\_1', 'A')

)

CALL spark\_catalog.system.add\_files(

table => 'db.tbl',

source\_table => '`parquet`.`path/to/table`'

)

**5）元数据信息**

（1）获取指定快照的父快照id

CALL spark\_catalog.system.ancestors\_of('db.tbl')

（2）获取指定快照的所有祖先快照

CALL spark\_catalog.system.ancestors\_of('db.tbl', 1)

CALL spark\_catalog.system.ancestors\_of(snapshot\_id => 1, table => 'db.tbl')

## 4.4 DataFrame 操作

### 4.4.1 环境准备

**1）创建maven工程，配置pom文件**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.atguigu.iceberg</groupId>

<artifactId>spark-iceberg-demo</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<scala.binary.version>2.12</scala.binary.version>

<spark.version>3.3.1</spark.version>

<maven.compiler.source>8</maven.compiler.source>

<maven.compiler.target>8</maven.compiler.target>

</properties>

<dependencies>

<!-- Spark的依赖引入 -->

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-core\_${scala.binary.version}</artifactId>

<scope>provided</scope>

<version>${spark.version}</version>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-sql\_${scala.binary.version}</artifactId>

<scope>provided</scope>

<version>${spark.version}</version>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-hive\_${scala.binary.version}</artifactId>

<scope>provided</scope>

<version>${spark.version}</version>

</dependency>

<!--fastjson <= 1.2.80 存在安全漏洞,-->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.83</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.iceberg/iceberg-spark-runtime-3.3 -->

<dependency>

<groupId>org.apache.iceberg</groupId>

<artifactId>iceberg-spark-runtime-3.3\_2.12</artifactId>

<version>1.1.0</version>

</dependency>

</dependencies>

<build>

<plugins>

<!-- assembly打包插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<version>3.0.0</version>

<executions>

<execution>

<id>make-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

<configuration>

<archive>

<manifest>

</manifest>

</archive>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</plugin>

<!--Maven编译scala所需依赖-->

<plugin>

<groupId>net.alchim31.maven</groupId>

<artifactId>scala-maven-plugin</artifactId>

<version>3.2.2</version>

<executions>

<execution>

<goals>

<goal>compile</goal>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

**2）配置Catalog**

val spark: SparkSession = SparkSession.builder().master("local").appName(this.getClass.getSimpleName)  
 //指定hive catalog, catalog名称为iceberg\_hive  
 .config("spark.sql.catalog.iceberg\_hive", "org.apache.iceberg.spark.SparkCatalog")  
 .config("spark.sql.catalog.iceberg\_hive.type", "hive")  
 .config("spark.sql.catalog.iceberg\_hive.uri", "thrift://hadoop1:9083")  
 // .config("iceberg.engine.hive.enabled", "true")  
 //指定hadoop catalog，catalog名称为iceberg\_hadoop   
 .config("spark.sql.catalog.iceberg\_hadoop", "org.apache.iceberg.spark.SparkCatalog")  
 .config("spark.sql.catalog.iceberg\_hadoop.type", "hadoop")  
 .config("spark.sql.catalog.iceberg\_hadoop.warehouse", "hdfs://hadoop1:8020/warehouse/spark-iceberg")  
 .getOrCreate()

### 4.4.2 读取表

**1）加载表**

spark.read

.format("iceberg")

.load("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a")

.show()

**或**

// 仅支持Spark3.0以上

spark.table("iceberg\_hadoop.default.a")

.show()

**2）时间旅行：指定时间查询**

spark.read

.option("as-of-timestamp", "499162860000")

.format("iceberg")

.load("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a")

.show()

**3）时间旅行：指定快照id查询**

spark.read

.option("snapshot-id", 7601163594701794741L)

.format("iceberg")

.load("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a")

.show()

**4）增量查询**

spark.read

.format("iceberg")

.option("start-snapshot-id", "10963874102873")

.option("end-snapshot-id", "63874143573109")

.load("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a")

.show()

查询的表只能是append的方式写数据，不支持replace, overwrite, delete操作。

### 4.4.3 检查表

**1）查询元数据**

spark.read.format("iceberg").load("iceberg\_hadoop.default.a.files")

spark.read.format("iceberg").load("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a#files")

**2）元数据表时间旅行查询**

spark.read

.format("iceberg")

.option("snapshot-id", 7601163594701794741L)

.load("iceberg\_hadoop.default.a.files")

### 4.4.4 写入表

**1）创建样例类，准备DF**

case class Sample(id:Int,data:String,category:String)

val df: DataFrame = spark.createDataFrame(Seq(Sample(1,'A', 'a'), Sample(2,'B', 'b'), Sample(3,'C', 'c')))

**2）插入数据并建表**

df.writeTo("iceberg\_hadoop.default.table1").create()

import spark.implicits.\_  
df.writeTo("iceberg\_hadoop.default.table1")  
 .tableProperty("write.format.default", "orc")  
 .partitionedBy($"category")  
 .createOrReplace()

**3）append追加**

df.writeTo("iceberg\_hadoop.default.table1").append()

**4）动态分区覆盖**

df.writeTo("iceberg\_hadoop.default.table1").overwritePartitions()

**5）静态分区覆盖**

import spark.implicits.\_

df.writeTo("iceberg\_hadoop.default.table1").overwrite($"category" === "c")

**6）插入分区表且分区内排序**

df.sortWithinPartitions("category")

.writeTo("iceberg\_hadoop.default.table1")

.append()

### 4.4.5 维护表

**1）获取Table对象**

（1）HadoopCatalog

import org.apache.hadoop.conf.Configuration;

import org.apache.iceberg.hadoop.HadoopCatalog;

import org.apache.iceberg.Table;

import org.apache.iceberg.catalog.TableIdentifier;

val conf = new Configuration()

val catalog = new HadoopCatalog(conf,"hdfs://hadoop1:8020/warehouse/spark-iceberg")

val table: Table = catalog.loadTable(TableIdentifier.of("db","table1"))

（2）HiveCatalog

import org.apache.iceberg.hive.HiveCatalog;

import org.apache.iceberg.Table;

import org.apache.iceberg.catalog.TableIdentifier;

val catalog = new HiveCatalog()

catalog.setConf(spark.sparkContext.hadoopConfiguration)

val properties = new util.HashMap[String,String]()

properties.put("warehouse", "hdfs://hadoop1:8020/warehouse/spark-iceberg")

properties.put("uri", "thrift://hadoop1:9083")

catalog.initialize("hive", properties)

val table: Table = catalog.loadTable(TableIdentifier.of("db", "table1"))

**2）快照过期清理**

每次写入Iceberg表都会创建一个表的新快照或版本。快照可以用于时间旅行查询，或者可以将表回滚到任何有效的快照。建议设置快照过期时间，过期的旧快照将从元数据中删除（不再可用于时间旅行查询）。

// 1天过期时间

val tsToExpire: Long = System.currentTimeMillis() - (1000 \* 60 \* 60 \* 24)

table.expireSnapshots()

.expireOlderThan(tsToExpire)

.commit()

或使用SparkActions来设置过期：

//SparkActions可以并行运行大型表的表过期设置

SparkActions.get()

.expireSnapshots(table)

.expireOlderThan(tsToExpire)

.execute()

**3）删除无效文件**

在Spark和其他分布式处理引擎中，任务或作业失败可能会留下未被表元数据引用的文件，在某些情况下，正常的快照过期可能无法确定不再需要并删除该文件。

SparkActions

.get()

.deleteOrphanFiles(table)

.execute()

**4）合并小文件**

数据文件过多会导致更多的元数据存储在清单文件中，而较小的数据文件会导致不必要的元数据量和更低效率的文件打开成本。

SparkActions

.get()

.rewriteDataFiles(table)

.filter(Expressions.equal("category", "a"))

.option("target-file-size-bytes", 1024L.toString) //1KB

.execute()

## 4.5 Merge On Read 与 Copy On Write

Iceberg当前支持有两种表读写类型Copy On Write（写复制）和Merge On Read（合并读）。这两种表类型决定数据如何写入、存储，索引和查询。

* Copy-On-Write数据在写入期间执行同步合并并重写文件，任务每次提交数据文件都会进行隐式的压缩合并。
* Merge on Read则不然，它在发生修改操作时，是将变化数据写入到增量变更的行操作文件。

Copy on Write表的更新数据写入延迟高，读取时消耗小。而Merge on Read表写入更新数据时成本低，而读取阶段消耗大。

我们通常可以采用任务写入时使用Merge on Read，然后定时启动后台异步任务的方式合并文件，从而减少读取时的性能消耗。在一定程序上达到一种平衡，从而取得更好的数据体验。

### 4.5.1 格式版本(format version)

最初的 Iceberg 规范(V1)描述了 Iceberg 目前存在的许多核心设计和行为。它概述了如何通过表元数据文件的原子交换实现乐观并发控制，并定义了读和写的要求。它还列出了兼容文件系统所需的最低限度操作，详细介绍了所有元数据文件、属性和数据类型，以及模式和分区等表结构。

随着涉及行级设计的出现，该特性需要对 Iceberg 表规范进行破坏性的更改，包括对 Iceberg 客户端和所有计算引擎进行广泛的更改，因此Iceberg定义了新版本的表规范V2。

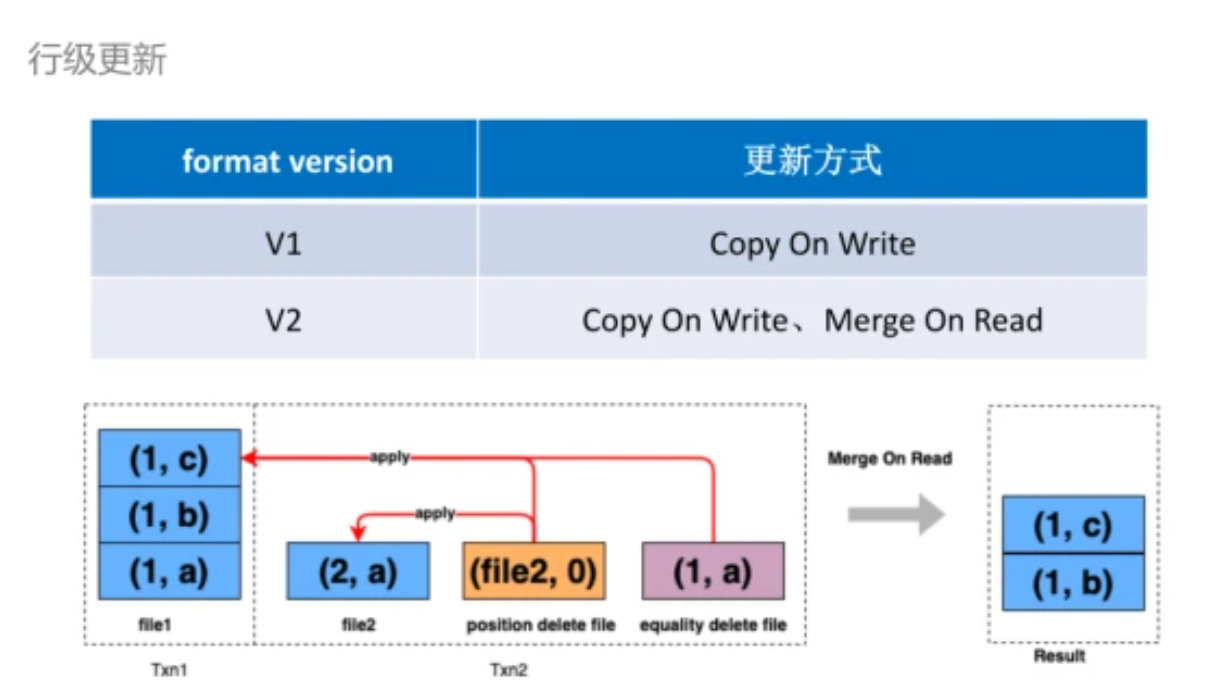
**1）版本1:分析数据表**

Iceberg第1版规范定义了如何使用不可变文件格式(Parquet、Avro和ORC)管理大型分析表。将表升级到版本2后，所有版本1的数据和元数据文件都有效。

**2）版本2:行级删除**

Iceberg第2版规范为具有不可变文件的分析表添加了行级更新和删除。

版本2主要增加了删除文件，以对现有数据文件中删除的行进行编码。此版本可用于删除或替换不可变数据文件中的单个行，而无需重写文件。



通过记录另外两个文件，即位置删除（position delete）和相等删除（equality delete）文件来对已有的文件进行删除，当读取的时候进行merge得到最终的结果。

### 4.5.2 相关参数

|  |  |  |
| --- | --- | --- |
| 配置项 | 默认值 | 说明 |
| format-version | 1 | 表格式版本(可以是1或2)，在表属性中定义 |
| write.delete.mode | copy-on-write | 删除模式，copy-on-write或merge-on-read，仅支持V2表格式版本 |
| write.update.mode | copy-on-write | 更新模式，copy-on-write或merge-on-read，仅支持V2表格式版本 |
| write.merge.mode | copy-on-write | 合并模式，copy-on-write或merge-on-read，仅支持V2表格式版本 |

要使用V2的行级更新和删除，需要配置扩展spark.sql.extensions：

### 4.5.3 COW 表的更新和删除

**1）建表**

CREATE TABLE hadoop\_prod.default.c1 (

id bigint,

count bigint

)

USING iceberg;

**2）插入数据**

INSERT INTO hadoop\_prod.default.c1 VALUES (1, 1), (2, 2), (3, 3);

**3）更新数据**

UPDATE hadoop\_prod.default.c1 SET count=4 WHERE id=3;

**4）删除数据**

DELETE FROM hadoop\_prod.default.c1 WHERE id=3;

**5）查看元数据文件**

打开最新的元数据json文件，可以分析出变化：

{

"format-version" : 1,

……

"snapshots" : [ {

……

"summary" : {

"operation" : "append", -- 第一次插入3条数据

"spark.app.id" : "local-1677834993216",

"added-data-files" : "2",

"added-records" : "3",

"added-files-size" : "1361",

"changed-partition-count" : "1",

"total-records" : "3",

"total-files-size" : "1361",

"total-data-files" : "2",

"total-delete-files" : "0",

"total-position-deletes" : "0",

"total-equality-deletes" : "0"

},

……

}, {

……

"summary" : {

"operation" : "overwrite", -- 更新id=3的一条数据

"spark.app.id" : "local-1677834993216",

"added-data-files" : "1",

"deleted-data-files" : "1",

"added-records" : "2",

"deleted-records" : "2",

"added-files-size" : "701",

"removed-files-size" : "683",

"changed-partition-count" : "1",

"total-records" : "3",

"total-files-size" : "1379",

"total-data-files" : "2",

"total-delete-files" : "0",

"total-position-deletes" : "0",

"total-equality-deletes" : "0"

},

……

}, {

……

"summary" : {

"operation" : "overwrite", -- 删除id=3的一条数据

"spark.app.id" : "local-1677834993216",

"added-data-files" : "1",

"deleted-data-files" : "1",

"added-records" : "1",

"deleted-records" : "2",

"added-files-size" : "693",

"removed-files-size" : "701",

"changed-partition-count" : "1",

"total-records" : "2",

"total-files-size" : "1371",

"total-data-files" : "2",

"total-delete-files" : "0",

"total-position-deletes" : "0",

"total-equality-deletes" : "0"

},

……

} ],

……

}

### 4.5.4 MOR 表的更新和删除

**1）配置扩展**

vim spark-default.conf

spark.sql.extensions =org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions

**2）建表**

CREATE TABLE hadoop\_prod.default.c2 (

id bigint,

count bigint

)

USING iceberg

TBLPROPERTIES(

'format-version'='2',

'write.delete.mode'='merge-on-read',

'write.update.mode'='merge-on-read',

'write.merge.mode'='merge-on-read'

)

**3）插入数据**

INSERT INTO hadoop\_prod.default.c2 VALUES (1, 1), (2, 2), (3, 3);

**4）更新数据**

UPDATE hadoop\_prod.default.c2 SET count=4 WHERE id=3;

**5）删除数据**

DELETE FROM hadoop\_prod.default.c2 WHERE id=3;

**6）查看元数据文件**

打开最新的元数据json文件，可以分析出变化：

{

"format-version" : 2,

……

"properties" : {

"owner" : "atguigu",

"write.merge.mode" : "merge-on-read",

"write.update.mode" : "merge-on-read",

"write.delete.mode" : "merge-on-read"

},

……

"snapshots" : [ {

"sequence-number" : 1,

……

"summary" : {

"operation" : "append", -- 第一次插入3条数据

"spark.app.id" : "local-1677834993216",

"added-data-files" : "2",

"added-records" : "3",

"added-files-size" : "1361",

"changed-partition-count" : "1",

"total-records" : "3",

"total-files-size" : "1361",

"total-data-files" : "2",

"total-delete-files" : "0",

"total-position-deletes" : "0",

"total-equality-deletes" : "0"

},

……

}, {

"sequence-number" : 2,

……

"summary" : {

"operation" : "overwrite", -- 更新id=3的一条数据

"spark.app.id" : "local-1677834993216",

"added-data-files" : "1",

"added-position-delete-files" : "1",

"added-delete-files" : "1",

"added-records" : "1",

"added-files-size" : "2230",

"added-position-deletes" : "1",

"changed-partition-count" : "1",

"total-records" : "4",

"total-files-size" : "3591",

"total-data-files" : "3",

"total-delete-files" : "1",

"total-position-deletes" : "1",

"total-equality-deletes" : "0"

},

……

}, {

"sequence-number" : 3,

……

"summary" : {

"operation" : "overwrite", -- 删除id=3的一条数据

"spark.app.id" : "local-1677834993216",

"added-position-delete-files" : "1",

"added-delete-files" : "1",

"added-files-size" : "1534",

"added-position-deletes" : "1",

"changed-partition-count" : "1",

"total-records" : "4",

"total-files-size" : "5125",

"total-data-files" : "3",

"total-delete-files" : "2",

"total-position-deletes" : "2",

"total-equality-deletes" : "0"

},

……

} ],

}

# 第5章 与Flink SQL集成

Apache Iceberg同时支持Apache Flink的DataStream API和Table API。

## 5.1 环境准备

### 5.1.1 安装 Flink

**1）Flink与Iceberg的版本对应关系如下**

|  |  |
| --- | --- |
| Flink 版本 | Iceberg 版本 |
| 1.11 | 0.9.0 – 0.12.1 |
| 1.12 | 0.12.0 – 0.13.1 |
| 1.13 | 0.13.0 – 1.0.0 |
| 1.14 | 0.13.0 – 1.1.0 |
| 1.15 | 0.14.0 – 1.1.0 |
| 1.16 | 1.1.0 – 1.1.0 |

**2）上传并解压Flink安装包**

tar -zxvf flink-1.16.0-bin-scala\_2.12.tgz -C /opt/module/

**3）配置环境变量**

sudo vim /etc/profile.d/my\_env.sh

export HADOOP\_CLASSPATH=`hadoop classpath`

source /etc/profile.d/my\_env.sh

**4）拷贝iceberg的jar包到Flink的lib目录**

cp /opt/software/iceberg/iceberg-flink-runtime-1.16-1.1.0.jar /opt/module/flink-1.16.0/lib

### 5.1.2 启动 Hadoop

（略）

### 5.1.3 启动 sql-client

**1）修改flink-conf.yaml配置**

vim /opt/module/flink-1.16.0/conf/flink-conf.yaml

classloader.check-leaked-classloader: false

taskmanager.numberOfTaskSlots: 4

state.backend: rocksdb

execution.checkpointing.interval: 30000

state.checkpoints.dir: hdfs://hadoop1:8020/ckps

state.backend.incremental: true

**2）local模式**

（1）修改workers

vim /opt/module/flink-1.16.0/conf/workers

#表示：会在本地启动3个TaskManager的 local集群

localhost

localhost

localhost

（2）启动Flink

/opt/module/flink-1.16.0/bin/start-cluster.sh

查看webui：<http://hadoop1:8081>

（3）启动Flink的sql-client

/opt/module/flink-1.16.0/bin/sql-client.sh embedded



## 5.2 创建和使用 Catalog

### 5.2.1 语法说明

CREATE CATALOG <catalog\_name> WITH (

'type'='iceberg',

`<config\_key>`=`<config\_value>`

);

* type: 必须是iceberg。（必须）
* catalog-type: 内置了hive和hadoop两种catalog，也可以使用catalog-impl来自定义catalog。（可选）
* catalog-impl: 自定义catalog实现的全限定类名。如果未设置catalog-type，则必须设置。（可选）
* property-version: 描述属性版本的版本号。此属性可用于向后兼容，以防属性格式更改。当前属性版本为1。（可选）
* cache-enabled: 是否启用目录缓存，默认值为true。（可选）
* cache.expiration-interval-ms: 本地缓存catalog条目的时间(以毫秒为单位)；负值，如-1表示没有时间限制，不允许设为0。默认值为-1。（可选）

### 5.2.2 Hive Catalog

**1）上传hive connector到flink的lib中**

cp flink-sql-connector-hive-3.1.2\_2.12-1.16.0.jar /opt/module/flink-1.16.0/lib/

**2）启动hive metastore服务**

hive --service metastore

**3）创建hive catalog**

重启flink集群，重新进入sql-client。

CREATE CATALOG hive\_catalog WITH (

'type'='iceberg',

'catalog-type'='hive',

'uri'='thrift://hadoop1:9083',

'clients'='5',

'property-version'='1',

'warehouse'='hdfs://hadoop1:8020/warehouse/iceberg-hive'

);

use catalog hive\_catalog;

* uri: Hive metastore的thrift uri。(必选)
* clients:Hive metastore客户端池大小，默认为2。(可选)
* warehouse: 数仓目录。
* hive-conf-dir:包含hive-site.xml配置文件的目录路径，hive-site.xml中hive.metastore.warehouse.dir 的值会被warehouse覆盖。
* hadoop-conf-dir:包含core-site.xml和hdfs-site.xml配置文件的目录路径。

### 5.2.3 Hadoop Catalog

Iceberg还支持HDFS中基于目录的catalog，可以使用'catalog-type'='hadoop'配置。

CREATE CATALOG hadoop\_catalog WITH (

'type'='iceberg',

'catalog-type'='hadoop',

'warehouse'='hdfs://hadoop1:8020/warehouse/iceberg-hadoop',

'property-version'='1'

);

use catalog hadoop\_catalog;

* warehouse:存放元数据文件和数据文件的HDFS目录。（必需）

### 5.2.4 配置sql-client初始化文件

vim /opt/module/flink-1.16.0/conf/sql-client-init.sql

CREATE CATALOG hive\_catalog WITH (

'type'='iceberg',

'catalog-type'='hive',

'uri'='thrift://hadoop1:9083',

'warehouse'='hdfs://hadoop1:8020/warehouse/iceberg-hive'

);

USE CATALOG hive\_catalog;

后续启动sql-client时，加上 -i sql文件路径 即可完成catalog的初始化。

/opt/module/flink-1.16.0/bin/sql-client.sh embedded -i conf/sql-client-init.sql

## 5.3 DDL 语句

### 5.3.1 创建数据库

CREATE DATABASE iceberg\_db;

USE iceberg\_db;

### 5.3.2 创建表

CREATE TABLE `hive\_catalog`.`default`.`sample` (

id BIGINT COMMENT 'unique id',

data STRING

);

建表命令现在支持最常用的flink建表语法，包括：

* PARTITION BY (column1, column2, ...)：配置分区，apache flink还不支持隐藏分区。
* COMMENT 'table document'：指定表的备注
* WITH ('key'='value', ...)：设置表属性

目前，不支持计算列、watermark（支持主键）。

**1）创建分区表**

CREATE TABLE `hive\_catalog`.`default`.`sample` (

id BIGINT COMMENT 'unique id',

data STRING

) PARTITIONED BY (data);

Apache Iceberg支持隐藏分区，但Apache flink不支持在列上通过函数进行分区，现在无法在flink DDL中支持隐藏分区。

**2）使用LIKE语法建表**

LIKE语法用于创建一个与另一个表具有相同schema、分区和属性的表。

CREATE TABLE `hive\_catalog`.`default`.`sample` (

id BIGINT COMMENT 'unique id',

data STRING

);

CREATE TABLE `hive\_catalog`.`default`.`sample\_like` LIKE `hive\_catalog`.`default`.`sample`;

### 5.3.3 修改表

**1）修改表属性**

ALTER TABLE `hive\_catalog`.`default`.`sample` SET ('write.format.default'='avro');

**2）修改表名**

ALTER TABLE `hive\_catalog`.`default`.`sample` RENAME TO `hive\_catalog`.`default`.`new\_sample`;

### 5.3.4 删除表

DROP TABLE `hive\_catalog`.`default`.`sample`;

## 5.4 插入语句

### 5.4.1 INSERT INTO

INSERT INTO `hive\_catalog`.`default`.`sample` VALUES (1, 'a');

INSERT INTO `hive\_catalog`.`default`.`sample` SELECT id, data from sample2;

### 5.4.2 INSERT OVERWRITE

仅支持Flink的Batch模式。

SET execution.runtime-mode = batch;

INSERT OVERWRITE sample VALUES (1, 'a');

INSERT OVERWRITE `hive\_catalog`.`default`.`sample` PARTITION(data='a') SELECT 6;

### 5.4.3 UPSERT

当将数据写入v2表格式时，Iceberg支持基于主键的UPSERT。有两种方法可以启用upsert。

**1）建表时指定**

CREATE TABLE `hive\_catalog`.`test1`.`sample5` (

`id` INT UNIQUE COMMENT 'unique id',

`data` STRING NOT NULL,

PRIMARY KEY(`id`) NOT ENFORCED

) with (

'format-version'='2',

'write.upsert.enabled'='true'

);

**2）插入时指定**

INSERT INTO tableName /\*+ OPTIONS('upsert-enabled'='true') \*/

...

插入的表，format-version需要为2。

OVERWRITE和UPSERT不能同时设置。在UPSERT模式下，如果对表进行分区，则分区字段必须也是主键。

**3）读取Kafka流，upsert插入到iceberg表中**

create table default\_catalog.default\_database.kafka(

id int,

data string

) with (

'connector' = 'kafka'

,'topic' = 'test111'

,'properties.zookeeper.connect' = 'hadoop1:2181'

,'properties.bootstrap.servers' = 'hadoop1:9092'

,'format' = 'json'

,'properties.group.id'='iceberg'

,'scan.startup.mode'='earliest-offset'

);

INSERT INTO hive\_catalog.test1.sample5 SELECT \* FROM default\_catalog.default\_database.kafka;

## 5.5 查询语句

Iceberg支持Flink的流式和批量读取。

### 5.5.1 Batch模式

SET execution.runtime-mode = batch;

select \* from sample;

### 5.5.2 Streaming模式

SET execution.runtime-mode = streaming;

SET table.dynamic-table-options.enabled=true;

SET sql-client.execution.result-mode=tableau;

**1）从当前快照读取所有记录，然后从该快照读取增量数据**

SELECT \* FROM sample5 /\*+ OPTIONS('streaming'='true', 'monitor-interval'='1s')\*/ ;

**2）读取指定快照id（不包含）后的增量数据**

SELECT \* FROM sample /\*+ OPTIONS('streaming'='true', 'monitor-interval'='1s', 'start-snapshot-id'='3821550127947089987')\*/ ;

* monitor-interval: 连续监控新提交数据文件的时间间隔（默认为10s）。
* start-snapshot-id: 流作业开始的快照id。

**注意：**如果是无界数据流式upsert进iceberg表（读kafka，upsert进iceberg表），那么再去流读iceberg表会存在读不出数据的问题。如果无界数据流式append进iceberg表（读kafka，append进iceberg表），那么流读该iceberg表可以正常看到结果。

## 5.6 与 Flink 集成的不足

|  |  |  |
| --- | --- | --- |
| 支持的特性 | Flink | 备注 |
| [SQL create catalog](https://iceberg.apache.org/docs/latest/flink/#creating-catalogs-and-using-catalogs) | √ |  |
| [SQL create database](https://iceberg.apache.org/docs/latest/flink/#create-database) | √ |  |
| [SQL create table](https://iceberg.apache.org/docs/latest/flink/#create-table) | √ |  |
| [SQL create table like](https://iceberg.apache.org/docs/latest/flink/#create-table-like) | √ |  |
| [SQL alter table](https://iceberg.apache.org/docs/latest/flink/#alter-table) | √ | 只支持修改表属性，不支持更改列和分区 |
| [SQL drop\_table](https://iceberg.apache.org/docs/latest/flink/#drop-table) | √ |  |
| [SQL select](https://iceberg.apache.org/docs/latest/flink/#querying-with-sql) | √ | 支持流式和批处理模式 |
| [SQL insert into](https://iceberg.apache.org/docs/latest/flink/#insert-into) | √ | 支持流式和批处理模式 |
| [SQL insert overwrite](https://iceberg.apache.org/docs/latest/flink/#insert-overwrite) | √ |  |
| [DataStream read](https://iceberg.apache.org/docs/latest/flink/#reading-with-datastream) | √ |  |
| [DataStream append](https://iceberg.apache.org/docs/latest/flink/#appending-data) | √ |  |
| [DataStream overwrite](https://iceberg.apache.org/docs/latest/flink/#overwrite-data) | √ |  |
| [Metadata tables](https://iceberg.apache.org/docs/latest/flink/#inspecting-tables) |  | 支持Java API，不支持Flink SQL（计划于1.2.0支持） |
| [Rewrite files action](https://iceberg.apache.org/docs/latest/flink/#rewrite-files-action) | √ |  |

* 不支持创建隐藏分区的Iceberg表。
* 不支持创建带有计算列的Iceberg表。
* 不支持创建带watermark的Iceberg表。
* 不支持添加列，删除列，重命名列，更改列。
* Iceberg目前不支持Flink SQL 查询表的元数据信息，需要使用Java API 实现，计划于1.2.0支持。

# 第6章 与 Flink DataStream 集成

## 6.1 环境准备

### 6.1.1 配置pom文件

新建Maven工程，pom文件配置如下：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.atguigu.iceberg</groupId>

<artifactId>flink-iceberg-demo</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<maven.compiler.source>8</maven.compiler.source>

<maven.compiler.target>8</maven.compiler.target>

<flink.version>1.16.0</flink.version>

<java.version>1.8</java.version>

<scala.binary.version>2.12</scala.binary.version>

<slf4j.version>1.7.30</slf4j.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-java</artifactId>

<version>${flink.version}</version>

<scope>provided</scope> <!--不会打包到依赖中，只参与编译，不参与运行 -->

</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-clients</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-table-planner -->

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-table-planner\_${scala.binary.version}</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-connector-files</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<!--idea运行时也有webui-->

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-runtime-web</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4j.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>${slf4j.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-to-slf4j</artifactId>

<version>2.14.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-statebackend-rocksdb</artifactId>

<version>${flink.version}</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>3.1.3</version>

<scope>provided</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.iceberg/iceberg-flink-runtime-1.16 -->

<dependency>

<groupId>org.apache.iceberg</groupId>

<artifactId>iceberg-flink-runtime-1.16</artifactId>

<version>1.1.0</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>3.2.4</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<artifactSet>

<excludes>

<exclude>com.google.code.findbugs:jsr305</exclude>

<exclude>org.slf4j:\*</exclude>

<exclude>log4j:\*</exclude>

<exclude>org.apache.hadoop:\*</exclude>

</excludes>

</artifactSet>

<filters>

<filter>

<!-- Do not copy the signatures in the META-INF folder.

Otherwise, this might cause SecurityExceptions when using the JAR. -->

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*.RSA</exclude>

</excludes>

</filter>

</filters>

<transformers combine.children="append">

<transformer

implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer">

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

### 6.1.2 配置log4j

resources目录下新建log4j.properties。

log4j.rootLogger=error,stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n

## 6.2 读取数据

### 6.2.1 常规Source写法

**1）Batch方式**

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a");

DataStream<RowData> batch = FlinkSource.forRowData()

.env(env)

.tableLoader(tableLoader)

.streaming(false)

.build();

batch.map(r -> Tuple2.of(r.getLong(0),r.getLong(1) ))

.returns(Types.TUPLE(Types.LONG,Types.LONG))

.print();

env.execute("Test Iceberg Read");

**2）Streaming方式**

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a");

DataStream<RowData> stream = FlinkSource.forRowData()

.env(env)

.tableLoader(tableLoader)

.streaming(true)

.startSnapshotId(3821550127947089987L)

.build();

stream.map(r -> Tuple2.of(r.getLong(0),r.getLong(1) ))

.returns(Types.TUPLE(Types.LONG,Types.LONG))

.print();

env.execute("Test Iceberg Read");

### 6.2.2 FLIP-27 Source写法

FLIP-27的目标是解决当前流式源接口（SourceFunction）中存在的几个问题/缺陷，同时统一批处理和流处理API之间的源接口。希望解决的缺陷有：

* 目前需要为批处理和流处理执行分别实现不同的接口。
* “work discovery”（切片、分区等）和实际“读取”数据的逻辑混合在SourceFunction接口和DataStream API中，导致实现复杂。
* 分区/分片/切片在接口中不是显式的。这使得很难以一种与源无关的方式实现某些功能，例如事件时间对齐、每个分区的水印、动态分片分配。例如，Kafka和Kinesis的消费者都支持获取每个分区的水印，但在Flink 1.8.1中只有Kinesis消费者支持事件时间对齐（有选择地从分片中读取，从而确保我们在事件时间上一同均匀推进）。
* 检查点锁是由SourceFunction拥有的。实现必须确保在锁下进行元素传递和状态更新。Flink没有办法优化处理该锁。在锁争用的情况下，某些线程可能无法获得锁（检查点线程）。这也妨碍了一种无锁的Actor/Mailbox 线程模型。
* 没有通用的构建方法，这意味着每个源都需要自己实现一个复杂的线程模型。这使得实现和测试新源变得困难，并为现有源的更新\维护增高了门槛。

FLIP-27的模型：

图示

描述已自动生成

**1）Batch方式**

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a");

IcebergSource<RowData> source1 = IcebergSource.forRowData()

.tableLoader(tableLoader)

.assignerFactory(new SimpleSplitAssignerFactory())

.build();

DataStream<RowData> batch = env.fromSource(

Source1,

WatermarkStrategy.noWatermarks(),

"My Iceberg Source",

TypeInformation.of(RowData.class));

batch.map(r -> Tuple2.of(r.getLong(0), r.getLong(1)))

.returns(Types.TUPLE(Types.LONG, Types.LONG))

.print();

env.execute("Test Iceberg Read");

**2）Streaming方式**

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a");

IcebergSource source2 = IcebergSource.forRowData()

.tableLoader(tableLoader)

.assignerFactory(new SimpleSplitAssignerFactory())

.streaming(true)

.streamingStartingStrategy(StreamingStartingStrategy.INCREMENTAL\_FROM\_LATEST\_SNAPSHOT)

.monitorInterval(Duration.ofSeconds(60))

.build();

DataStream<RowData> stream = env.fromSource(

Source2,

WatermarkStrategy.noWatermarks(),

"My Iceberg Source",

TypeInformation.of(RowData.class));

stream.map(r -> Tuple2.of(r.getLong(0), r.getLong(1)))

.returns(Types.TUPLE(Types.LONG, Types.LONG))

.print();

env.execute("Test Iceberg Read");

## 6.3 写入数据

目前支持DataStream<RowData>和DataStream<Row>格式的数据流写入Iceberg表。

**1）写入方式支持 append、overwrite、upsert**

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setParallelism(1);

SingleOutputStreamOperator<RowData> input = env.fromElements("")

.map(new MapFunction<String, RowData>() {

@Override

public RowData map(String s) throws Exception {

GenericRowData genericRowData = new GenericRowData(2);

genericRowData.setField(0, 99L);

genericRowData.setField(1, 99L);

return genericRowData;

}

});

TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://hadoop1:8020/warehouse/spark-iceberg/default/a");

FlinkSink.forRowData(input)

.tableLoader(tableLoader)

.append() // append方式

//.overwrite(true) // overwrite方式

//.upsert(true) // upsert方式

;

env.execute("Test Iceberg DataStream");

**2）写入选项**

FlinkSink.forRowData(input)

.tableLoader(tableLoader)

.set("write-format", "orc")

.set(FlinkWriteOptions.OVERWRITE\_MODE, "true");

可配置选项如下：

|  |  |  |
| --- | --- | --- |
| 选项 | 默认值 | 说明 |
| write-format | Parquet  同write.format.default | 写入操作使用的文件格式：Parquet, avro或orc |
| target-file-size-bytes | 536870912（512MB）  同write.target-file-size-bytes | 控制生成的文件的大小，目标大约为这么多字节 |
| upsert-enabled | 同write.upsert.enabled， |  |
| overwrite-enabled | false | 覆盖表的数据，不能和UPSERT模式同时开启 |
| distribution-mode | None  同 write.distribution-mode | 定义写数据的分布方式:   * none:不打乱行; * hash:按分区键散列分布; * range：如果表有SortOrder，则通过分区键或排序键分配 |
| compression-codec | 同 write.(fileformat).compression-codec |  |
| compression-level | 同 write.(fileformat).compression-level |  |
| compression-strategy | 同write.orc.compression-strategy |  |

## 6.4 合并小文件

Iceberg现在不支持在flink sql中检查表，需要使用Iceberg提供的Java API来读取元数据来获得表信息。可以通过提交Flink批处理作业将小文件重写为大文件：

import org.apache.iceberg.flink.actions.Actions;

// 1.获取 Table对象

// 1.1 创建 catalog对象

Configuration conf = new Configuration();

HadoopCatalog hadoopCatalog = new HadoopCatalog(conf, "hdfs://hadoop1:8020/warehouse/spark-iceberg");

// 1.2 通过 catalog加载 Table对象

Table table = hadoopCatalog.loadTable(TableIdentifier.of("default", "a"));

// 有Table对象，就可以获取元数据、进行维护表的操作

// System.out.println(table.history());

// System.out.println(table.expireSnapshots().expireOlderThan());

// 2.通过 Actions 来操作 合并

Actions.forTable(table)

.rewriteDataFiles()

.targetSizeInBytes(1024L)

.execute();

得到Table对象，就可以获取元数据、进行维护表的操作。更多Iceberg提供的API操作，参考：<https://iceberg.apache.org/docs/latest/api/>