# 背景

随着互联网业务的逐步成熟，数仓和模型训练的基本盘逐步稳固，越来越多的工程师从业务开发需求转移到了工程的架构升级，而常用的 Hudi 和 Iceberg 往往会成为替代 Hive/Hdfs 等架构升级的选型。

先说背景，Hudi（Hadoop Upsert anD Incremental），从 Uber 内部孵化出来的开源项目，最初是用于解决数仓中 Lambda 架构中数据一致性的问题，将增量处理模型替代流式处理模型，并提供了 Upsert 和 Incremental Pull 两个非常重要的 feature。

这里可以提一下，Hudi 内部主打的一个场景，就是乘客打车下单和司机接单的匹配，乘客和司机分别是两条数据流，通过 Hudi 的 Upsert 能力和增量读取功能，可以分钟级地讲这两条数据流进行拼接，得到乘客-司机的匹配数据。

说明：

还记得两年前 Hudi 对自身的定位是一个 Pipeline 或者存储框架，现在官网的描述已经变成了「下一代的流式数据湖平台」，继商业化之后，想象空间也上升了好几个 Level。

# 概述

Apache Hudi依赖 HDFS 做底层的存储，所以可以支撑非常大规模的数据存储。同时基于下面两个原语，Hudi可以解决流批一体的存储问题。

提供了在Hadoop兼容的存储之上存储大量数据，同时它还提供两种原语：

Update/Delete 记录：Hudi 支持更新/删除记录，使用文件/记录级别索引，同时对写操作提供事务保证。查询可获取最新提交的快照来产生结果。

变更流：支持增量获取表中所有更新/插入/删除的记录，从指定时间点开始进行增量查询，可以实现类似 Kafka 的增量消费机制。

参考：

<http://shiyanjun.cn/archives/2043.html>

<https://zhuanlan.zhihu.com/p/464787222>

<http://www.liaojiayi.com/lake-hudi/>

## 设计原则

流式读/写：Hudi借鉴了数据库设计的原理，从零设计，应用于大型数据集记录流的输入和输出。为此，Hudi提供了索引实现，可以将记录的键快速映射到其所在的文件位置。同样，对于流式输出数据，Hudi通过其特殊列添加并跟踪记录级的元数据，从而可以提供所有发生变更的精确增量流。

自管理：Hudi注意到用户可能对数据新鲜度（写友好）与查询性能（读/查询友好）有不同的期望，它支持了三种查询类型，这些类型提供实时快照，增量流以及稍早的纯列数据。在每一步，Hudi都努力做到自我管理（例如自动优化编写程序的并行性，保持文件大小）和自我修复（例如：自动回滚失败的提交），即使这样做会稍微增加运行时成本（例如：在内存中缓存输入数据已分析工作负载）。如果没有这些内置的操作杠杆/自我管理功能，这些大型流水线的运营成本通常会翻倍。

万物皆日志：Hudi还具有 append only、云数据友好的设计，该设计实现了日志结构化存储系统的原理，可以无缝管理所有云提供商的数据。

键-值数据模型：在写方面，Hudi表被建模为键值对数据集，其中每条记录都有一个唯一的记录键。此外，一个记录键还可以包括分区路径，在该路径下，可以对记录进行分区和存储。这通常有助于减少索引查询的搜索空间。

## 基本概念

### Timeline

Timeline 可以理解为 Hudi 表的一个时间线，记录了 Hudi 表在不同时刻的信息和行为，这个 Timeline 由 TimelineServer 来管理，通常存在于 Hdfs、RDBMS 等持久化存储介质中。实际上，Hudi 将 Timeline 信息放到每个 Table 内的 .hoodie 目录中，并通过文件名来进行不同 instant 的区分。通过 Timeline 可以方便地做版本管理以及实现增量处理等和版本/时间相关的功能。

Timeline 涉及到 3 个关键概念：

action: 当前时刻的动作（类似 commit、rollback 等）

time: 当前时刻的时间点，毫秒级别

state: 当前动作的状态

可以看到，所有需要更改表元信息的操作，都是需要将对应的 action 提交至 Timeline，而 Timeline 的操作要保证原子性，一般要由单点进行操作，比如 Hudi 在与 Spark/Flink 结合时，利用 Spark 的 Driver 和 Flink 的 JobMaster 来进行 Timeline 的信息记录。

### Table Types & Query Types

Hudi 提供了两种表类型，分别为 Copy-on-Write 和 Merge-on-Read，其对应的查询类型如下：

对于 Copy-On-Write Table，用户的 update 会重写数据所在的文件，所以是一个写放大很高，但是读放大为 0，适合写少读多的场景。对于这种 Table，提供了两种查询：

Snapshot Query: 查询最近一次 snapshot 的数据，也就是最新的数据。

Incrementabl Query:用户需要指定一个 commit time，然后 Hudi 会扫描文件中的记录，过滤出 commit\_time > 用户指定的 commit time 的记录。

具体的流程见下图 gif:

对于 Merge-On-Read Table，整体的结构有点像 LSM-Tree，用户的写入先写入到 delta data 中，这部分数据使用行存，这部分 delta data 可以手动 merge 到存量文件中，整理为 parquet 的列存结构。对于这类 Table，提供了三种查询：

Snapshot Query: 查询最近一次 snapshot 的数据，也就是最新的数据。这里是一个行列数据混合的查询。

Incrementabl Query:用户需要指定一个 commit time，然后 Hudi 会扫描文件中的记录，过滤出 commit\_time > 用户指定的 commit time 的记录。这里是一个行列数据混合的查询。

Read Optimized Query: 只查存量数据，不查增量数据，因为使用的都是列式文件格式，所以效率较高。

具体的流程见下图 gif:

MOR 表中可能存在两种文件，在 Hudi 内部被称为 base file 和 log file，其中 base file 通常为 parquet 文件，列存格式，对读取友好，log file 通常为 avro 格式，行存，对写入友好。

### Index

Hudi 表有主键的概念，所以 Index 的出现也非常合理，可以用于定位数据的位置以提供更高效的写入和读取操作，不同的 Index 类型提供了不同的粒度：

Bloom Index

Simple Index

HBase Index

Hash Index

对于每条 Record，我们会查询/计算 Record 的主键所在索引的方式，来判断是 Insert 还是 Update，以及对应的旧文件的位置。在实时写入的过程中，Index 的查询是最关键的部分之一，索引设计的高效与否直接决定了数据写入的性能和稳定性。

### File Layouts

示意图如下所示，由外到里分别是：

Table

Partition

FileGroup(由 FileGroupId 或者 FileID 作为标识符)：Partition 由多个 FileGroup 组成，每个 FileGroup 包括一个 base file 和多个 log file

base file 和 log file：详见上面对 MOR 表的阐述

# 核心功能

## Upsert

上面提到，Upsert 的操作和 Index 类型很相关，但是在 Hudi 内部有趣的是，由于初始架构设计的缺陷（并未考虑非 Spark 场景），导致了不同 Connector 在使用 Index 上有非常大的差异。

Hudi 目前支持 Spark 和 Flink，而我们这里也以这两种计算引擎为例，讲解一下 Upsert 具体的实现机制：

* Spark：

对重复 PK 的数据进行 dedup，如果 Payload 实现了 preCombine 方法，对于相同 PK 的数据会调用 preCombine 进行 Payload 的合并；如果没有实现，则使用 Hudi 遍历时相同 PK 的第一条数据

调用 Index 的 tagLocation 方法，根据 PK 查询到表中已存在数据的位置，并记录 fileId 和 commitTs（用于定位具体的位置），对于没有找到的数据，暂时将 location 置为空

根据 Partition 进行计数，计算每个 Partition 需要写入的数据条数，并生成对应的 WorkloadStat（即每个 Partition 对应的 insert 和 update 数量），并根据已有的文件分布（比如优先把新数据写入数据量小的 FileGroup）决定每个 FileGroup 写入的数据条数

给每条数据分配 FileGroup 位置，对于 UPDATE 数据，直接使用 Index 中获取的位置；对于 INSERT 数据，基于上一步骤的结果进行随机分配；将数据按照不同的 FileGroup 位置进行 partitionBy 操作，使得 Spark 的每个 Partition 都只处理一个 FileGroup 的数据

将 Spark partition 的数据进行写入，写入成功后返回每条数据的 location，并逐条更新 Index，更新成功后将此次数据写入 commit 到 Hudi 的 Timeline

* Flink：

作业开始一个新的 Checkpoint 后，相对应的会开启一个 Hudi 的新 instant，两次成功的 checkpoint 之间就是一次 Hudi 的 instant 数据写入事件

将数据按 PK 进行 keyBy 操作，保证相同 PK 数据都在同样的 Task 中被处理

利用 Flink State 来存储每个 PK 对应的位置，对于 UPDATE 数据，直接通过访问 State 来获取对应的 FileGroup 位置；对于 INSERT 数据，和 Spark 类似，通过已有的文件分布信息来决定插入的位置

根据 FileGroup Id 进行 keyBy 操作，保证相同位置的数据都在同样的 Task 中被处理

写入数据的 Task 以 batch size 的形式缓存和写入数据，并在 Checkpoint 时将写入成功的数据元信息发送给 JobMaster 进行 commit

## Bulk Insert

Bulk Insert 的操作比较简单，只用于某个 Partition 或者某个 Table 初次初始化时使用，由于没有 Update 操作，所以只需要考虑 Insert 情况，性能相比 Upsert 有非常大的提升。

## 增量读取

这里可以看一下 Hudi 官方的 Incremental Query 的示例

// spark-shell

// reload data

spark.

read.

format("hudi").

load(basePath).

createOrReplaceTempView("hudi\_trips\_snapshot")

val commits = spark.sql("select distinct(\_hoodie\_commit\_time) as commitTime from hudi\_trips\_snapshot order by commitTime").map(k => k.getString(0)).take(50)

val beginTime = commits(commits.length - 2) // commit time we are interested in

// incrementally query data

val tripsIncrementalDF = spark.read.format("hudi").

option(QUERY\_TYPE\_OPT\_KEY, QUERY\_TYPE\_INCREMENTAL\_OPT\_VAL).

option(BEGIN\_INSTANTTIME\_OPT\_KEY, beginTime).

load(basePath)

tripsIncrementalDF.createOrReplaceTempView("hudi\_trips\_incremental")

spark.sql("select `\_hoodie\_commit\_time`, fare, begin\_lon, begin\_lat, ts from hudi\_trips\_incremental where fare > 20.0").show()

每个 Hudi 表中都有一个隐藏列叫 \_hoodie\_commit\_time，类型是时间戳，表示数据 commit 的时间，在增量读取时，我们通过指定 commit 的时间戳进行增量的范围定义。在 Hudi 内部，通过用户指定的 beginTime 时间戳，对 Timeline 上的 Instant 进行筛选，得到 beginTime 后续的 Instant 范围，并找到对应的数据写入的 Metadata，来得到需要进行扫描的文件。如果遇到 Compaction 的情况，则在扫描时会对 Compaction 后的数据明细进行时间范围的过滤。

## 其他

事务支持/并发控制：最开始 Hudi 只支持 Table 级别的隔离，比如同时对同一个 Table 产生 commits，那么只有一个 commit 会成功，另一个 commit 会提交失败；后续做了一定的改进，将 Table 级别的隔离做到的 FileGroup 级别。这个在数据库系统中也是非常常见的做法。

preCombine 能力：HoodieRecordPayload 的实现如下，每条记录会被封装成一个 Payload，用户可以在 Payload 中实现相同 PK 的数据 merge 操作，默认是取最新的一条记录。

public interface HoodieRecordPayload<T extends HoodieRecordPayload> extends Serializable {

T preCombine(T oldValue);

}

# 架构

# 安装部署

参考：

<https://developer.aliyun.com/article/1272258>