大家好，我是土哥。

在 Flink 实时流中，经常会通过 Flink CDC 插件读取 Mysql 数据，然后写入 Hudi 中。所以在执行上述操作时，需要了解 Hudi 的基本概念以及操作原理，这样在近实时往 Hudi 中写数据时，遇到报错问题，才能及时处理。

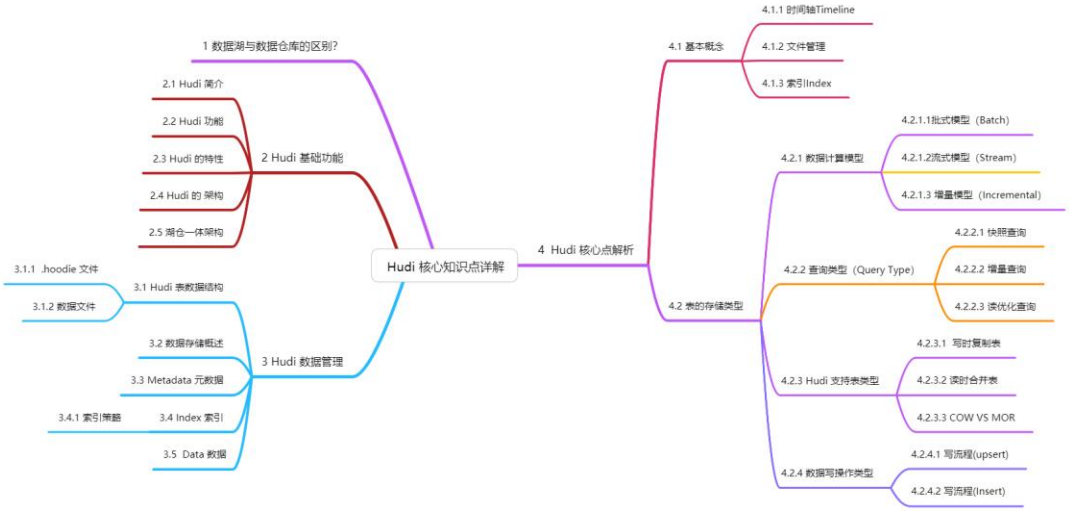
接下来将从以下几方面全面阐述 Hudi 组件核心知识点。

1.数据湖与数据仓库的区别？

2.Hudi 基础功能

3 Hudi 数据管理

4 Hudi 核心点解析



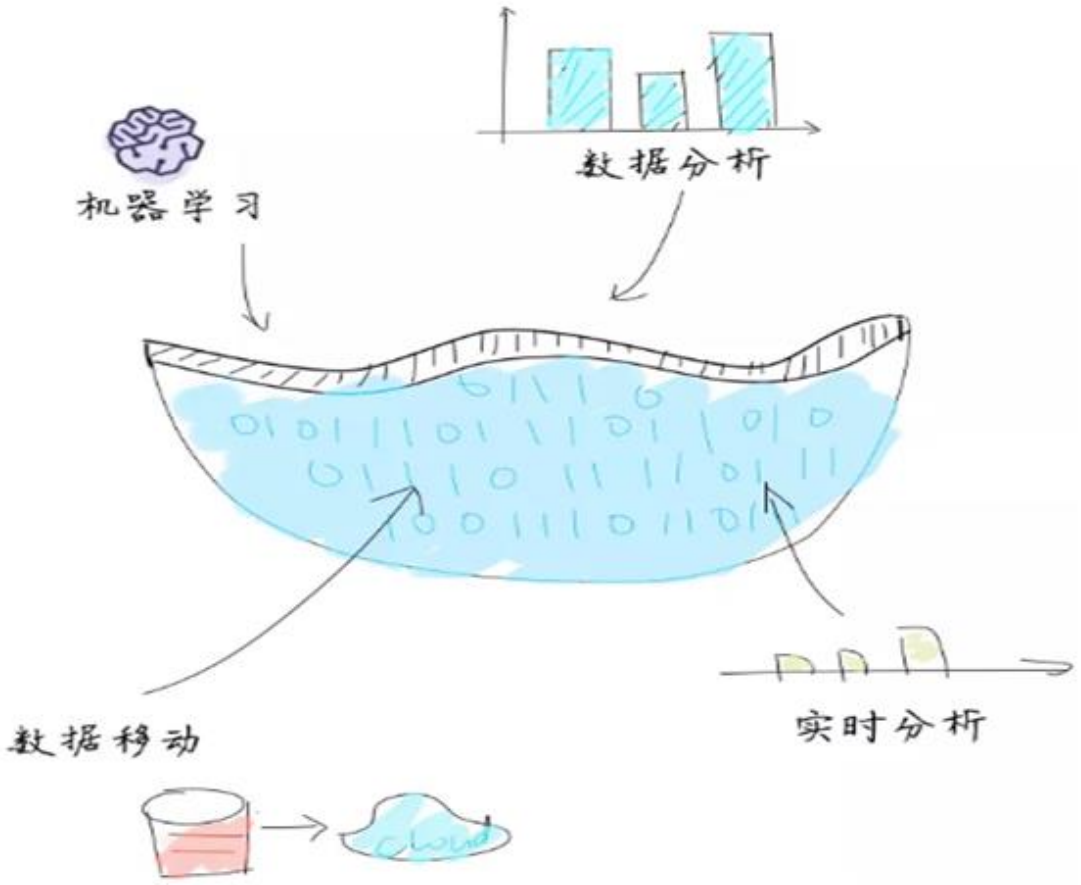
1 数据湖与数据仓库的区别？

数据仓库

- 数据仓库（英语：Data Warehouse，简称数仓、DW）,是一个用于**存储、分析、报告的数据系统**。
- 数据仓库的目的是构建**面向分析**的集成化数据环境，分析结果**为企业提供决策支持（Decision Support）**。

数据湖

- 数据湖（Data Lake）和数据库、数据仓库一样，都是**数据存储**的**设计模式，**现在企业的数据仓库都会通过**分层的方式**将数据存储在文件夹、文件中。
- 数据湖是一个**集中式**数据存储库，**用来存储**大量的原始数据，**使用**平面架构来存储数据。
- 定义：一个以**原始格式**(通常是对象块或文件)存储数据的**系统或存储库**，通常是所有企业数据的**单一存储**。
- 数据湖可以包括来自关系数据库的结构化数据(行和列)、半结构化数据(CSV、日志、XML、JSON)、非结构化数据(电子邮件、文档、pdf)和二进制数据(图像、音频、视频)。
- 数据湖中数据，用于报告、可视化、高级分析和机器学习等任务。



两者的区别：

- 数据仓库是一个优化的数据库，用于分析来自事务系统和业务线应用程序的关系数据。
- 数据湖存储来自业务线应用程序的关系数据，以及来自移动应用程序、IoT 设备和社交媒体的非关系数据。

特性	数据仓库	数据湖
数据	来自事务系统、运营数据库和业务线应用程序的关系数据	来自 IoT 设备、网站、移动应用程序、社交媒体和企业应用程序的非关系和关系数据
Schema	设计在数据仓库实施之前（写入型 Schema）	写入在分析时（读取型 Schema）
性价比	更快查询结果会带来较高存储成本	更快查询结果只需较低存储成本
数据质量	可作为重要事实依据的高度监管数据	任何可以或无法进行监管的数据（例如原始数据）
用户	业务分析师	数据科学家、数据开发人员和业务分析师（使用监管数据）
分析	批处理报告、BI 和可视化	机器学习、预测分析、数据发现和分析

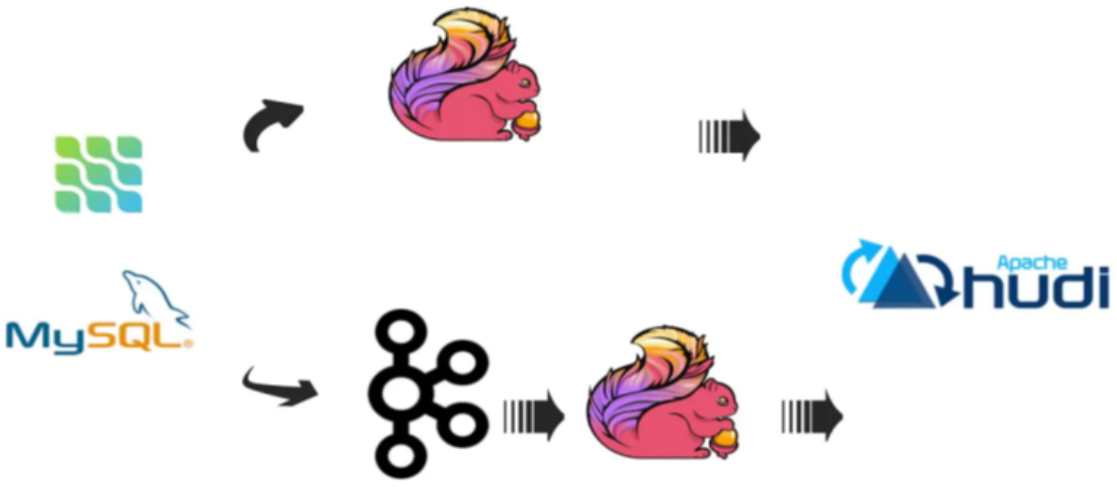
- **数据湖并不能替代数据仓库，数据仓库在高效的报表和可视化分析中仍有优势。**

2 Hudi 基础功能

2.1 Hudi 简介

Apache Hudi 由 Uber 开发并开源，该项目在 2016 年开始开发，并于 2017 年开源，2019 年 1 月进入 Apache 孵化器，且 2020 年 6 月称为 Apache 顶级项目，目前最新版本：0.10.1 版本。

Hudi 一开始支持 Spark 进行数据摄入（批量 Batch 和流式 Streaming），从 0.7.0 版本开始，逐渐与 Flink 整合，主要在于 Flink SQL 整合，还支持 Flink SQL CDC。



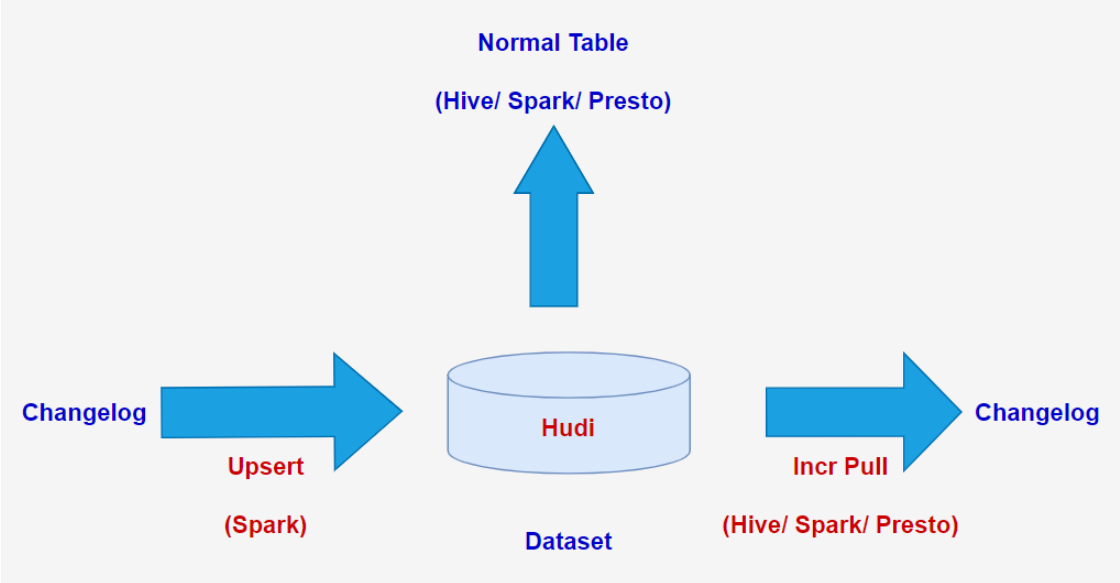
Hudi（**H**adoop **U**pserts an**D** **I**ncrementals 缩写）是目前市面上流行的三大开源数据湖方案之一。

用于管理分布式文件系统 DFS 上大型分析**数据集存储**。

简单来说，Hudi 是一种针对分析型业务的、扫描优化的**数据存储抽象**，它能够使 DFS 数据集在**分钟级的时延内支持变更**，也支持下游系统对这个数据集的增量处理。

2.2 Hudi 功能

- Hudi 是在大数据存储上的一个数据集，可以将 Change Logs 通过 **upsert** 的方式合并进 Hudi；
- Hudi 对上可以暴露成一个普通 Hive 或 Spark 表，通过 API 或命令行可以获取到增量修改的信息，继续供下游消费；
- Hudi 保管修改历史，可以做时间旅行或回退；
- Hudi 内部有**主键到文件级的索引**，默认是记录到文件的**布隆过滤器**；



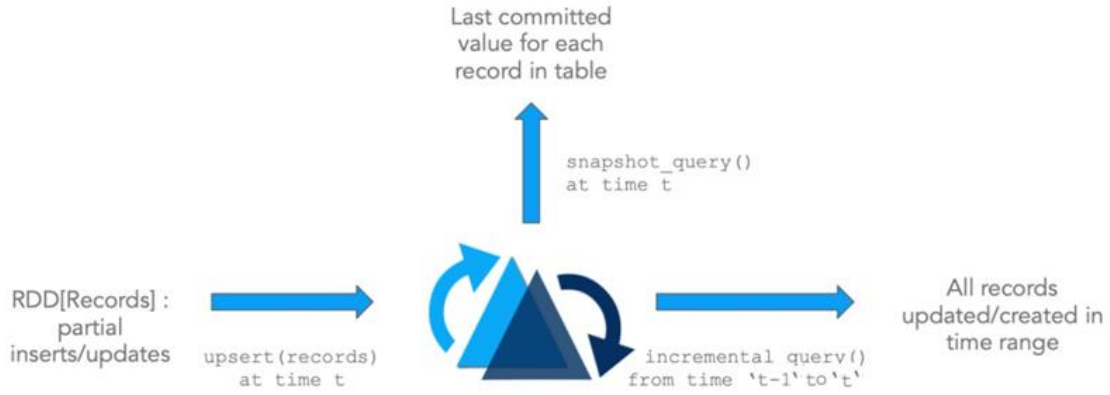
2.3 Hudi 的特性

Apache Hudi 使得用户能在 Hadoop 兼容的存储之上存储大量数据，同时它还提供两种原语，**不仅可以批处理**，还可以在数据湖上进行**流处理**。

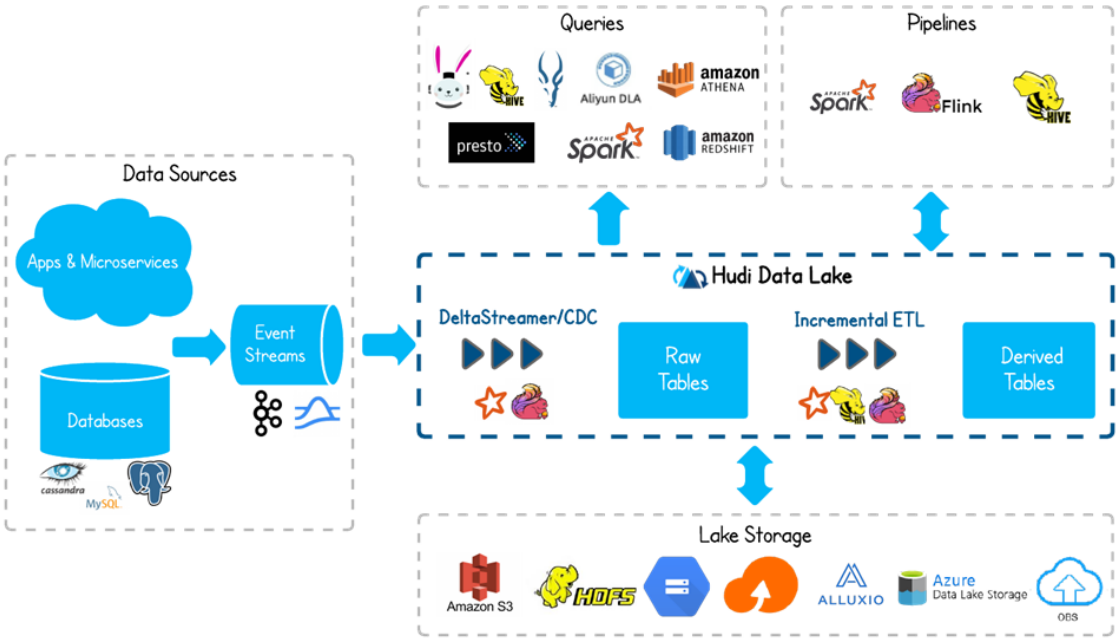
1. **Update/Delete 记录：**Hudi 使用细粒度的文件/记录级别索引来支持 Update/Delete 记录，**同时还提供写操作的事务保证**。查询会处理最后一个提交的快照，并基于此输出结果。

2. **变更流：**Hudi 对获取数据变更提供了一流的支持：可以从给定的 **时间点** 获取给定表中已 updated / inserted / deleted 的所有记录的增量流，并解锁新的查询姿势（类别）。

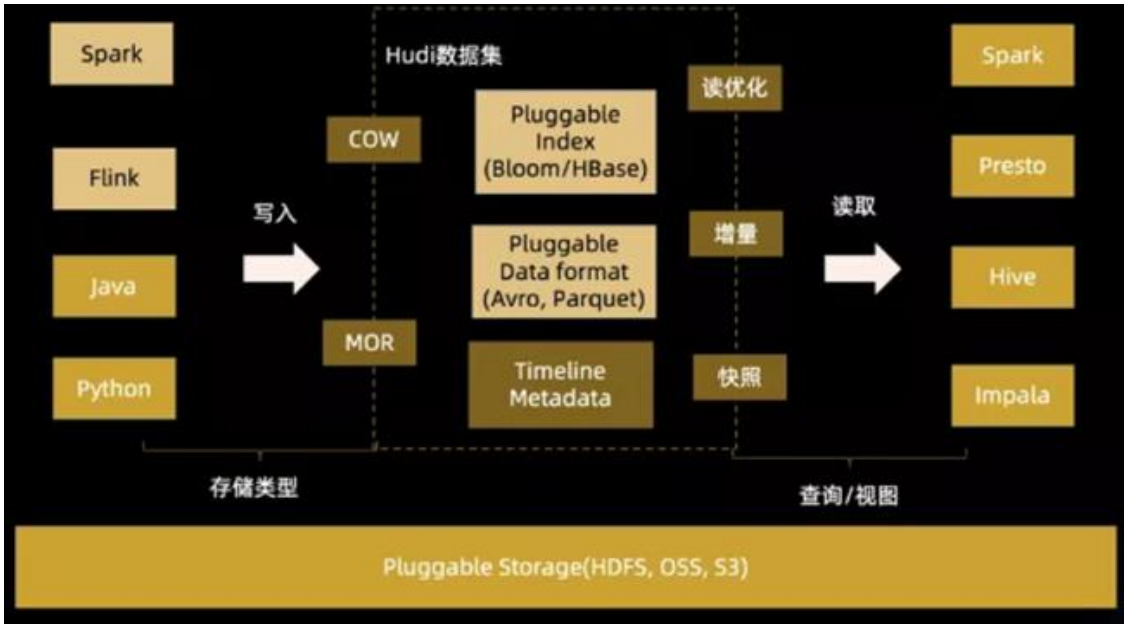
- Apache Hudi 本身不存储数据，仅仅管理数据。
- Apache Hudi 也不分析数据，需要使用计算分析引擎，查询和保存数据，比如 Spark 或 Flink；
- 使用 Hudi 时，加载 jar 包，底层调用 API，所以需要依据使用大数据框架版本，编译 Hudi 源码，获取对应依赖 jar 包。



2.4 Hudi 的 架构



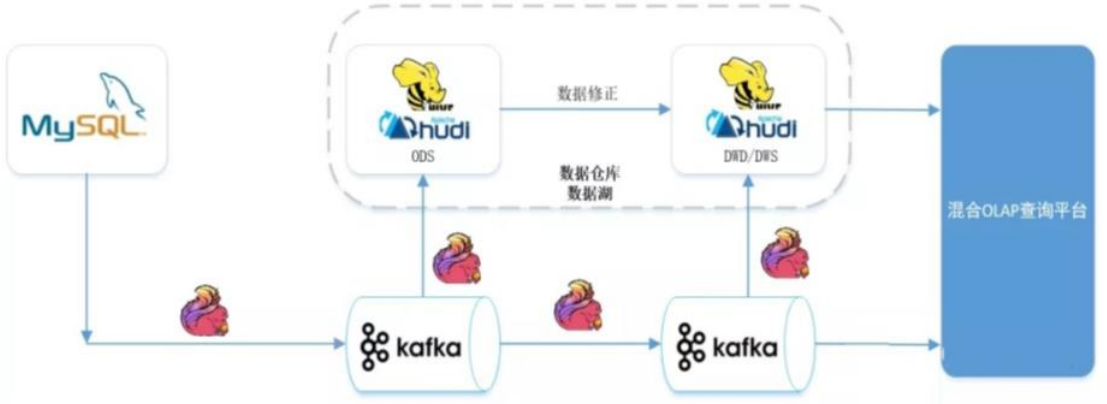
1. 通过 DeltaStreammer、Flink、Spark 等工具，将数据摄取到数据湖存储，可使用 HDFS 作为数据湖的数据存储；
2. 基于 HDFS 可以构建 Hudi 的数据湖；
3. Hudi 提供统一的访问 Spark 数据源和 Flink 数据源；
4. 外部通过不同引擎，如：Spark、Flink、Presto、Hive、Impala、Aliyun DLA、AWS Redshit 访问接口；



2.5 湖仓一体架构

Hudi 对于 Flink 友好支持以后，可以使用 Flink + Hudi 构建**实时湖仓一体**架构，数据的时效性可以到**分钟级**，能很好的满足业务**准实时数仓**的需求。

通过湖仓一体、流批一体，准实时场景下做到了：数据同源、同计算引擎、同存储、同计算口径。



3 Hudi 数据管理

3.1 Hudi 表数据结构

Hudi 表的数据文件，可以使用操作系统的文件系统存储，也可以使用 HDFS 这种分布式的文件系统存储。为了后续分析性能和数据的可靠性，一般**使用 HDFS 进行存储**。以 HDFS 存储来看，一个 Hudi 表的存储文件分为两类。

```
root@node1 ~]# hdfs dfs -ls /datas/hudi-warehouse/hudi_trips_cow
Found 3 items
drwxr-xr-x   - root supergroup          0 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie
drwxr-xr-x   - root supergroup          0 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/americas
drwxr-xr-x   - root supergroup          0 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/asia
```

1. **.hoodie 文件**：由于 CRUD 的零散性，每一次的操作都会生成一个文件，这些小文件越来越多后，会严重影响 HDFS 的性能，Hudi 设计了一套**文件合并机制**。 .hoodie 文件夹中存放了对应的 **文件合并操作** 相关的日志文件。
2. amricas 和 asia 相关的路径是 **实际的数据文件**，按分区存储，分区的路径 key 是可以指定的。

3.1.1 .hoodie 文件

Hudi 把随着时间流逝，对表的一系列 CRUD 操作叫做 **Timeline**，Timeline 中某一次的操作，叫做 **Instant**。

Hudi 的核心是维护 **Timeline** 在不同时间对表执行的所有操作，instant 这有助于提供表的即时视图，同时还有效地支持按到达顺序检索数据。Hudi Instant 由以下组件组成：

- Instant Action: 记录本次操作是一次操作类型 数据提交（COMMITS），还是文件合并（COMPACTION），或者是文件清理（CLEANS）；
- Instant Time，本次操作发生的时间，通常是时间戳（例如：20190117010349），它按照动作开始时间的顺序单调递增。
- lState，操作的状态，发起(REQUESTED)，进行中(INFLIGHT)，还是已完成(COMPLETED)；

.hoodie 文件夹中存放对应操作的状态记录：

```
[root@node1 ~]# hdfs dfs -ls /datas/hudi-warehouse/hudi_trips_cow/.hoodie
Found 7 items
drwxr-xr-x   - root supergroup          0 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie/.aux
drwxr-xr-x   - root supergroup          0 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie/.temp
-rw-r--r--   1 root supergroup       4187 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie/20210523174441.commit
-rw-r--r--   1 root supergroup          0 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie/20210523174441.commit.requested
-rw-r--r--   1 root supergroup       2594 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie/20210523174441.inflight
drwxr-xr-x   - root supergroup          0 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie/archived
-rw-r--r--   1 root supergroup        268 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/.hoodie/hoodie.properties
```

3.1.2 数据文件

Hudi 真实的数据文件使用 Parquet 文件格式存储

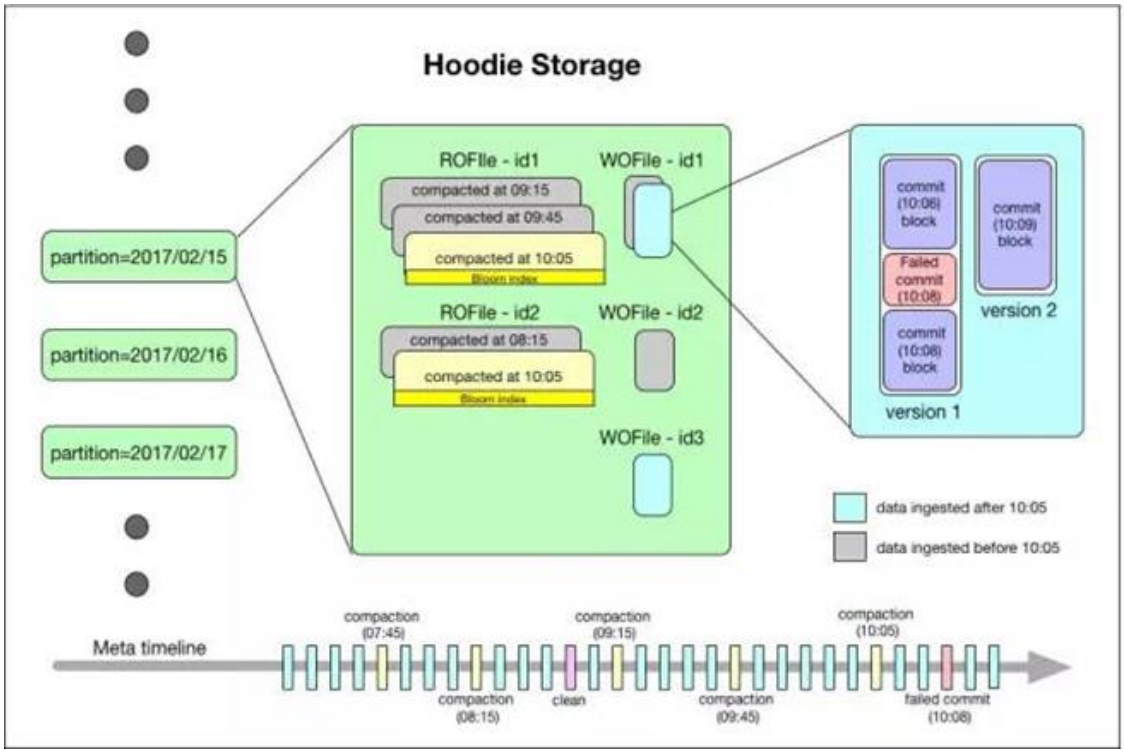
```
[root@node1 ~]# hdfs dfs -ls /datas/hudi-warehouse/hudi_trips_cow/asia/india/chennai
Found 2 items
-rw-r--r--   1 root supergroup         93 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/asia/india/chennai/.hoodie_partition_metadata
-rw-r--r--   1 root supergroup     437200 2021-05-23 17:44 /datas/hudi-warehouse/hudi_trips_cow/asia/india/chennai/fbd5f23d-891e-475e-a4fd-f482d95bf
cfb-0_2-57-72_20210523174441.parquet
```

其中包含一个 metadata 元数据文件和数据文件 parquet 列式存储。

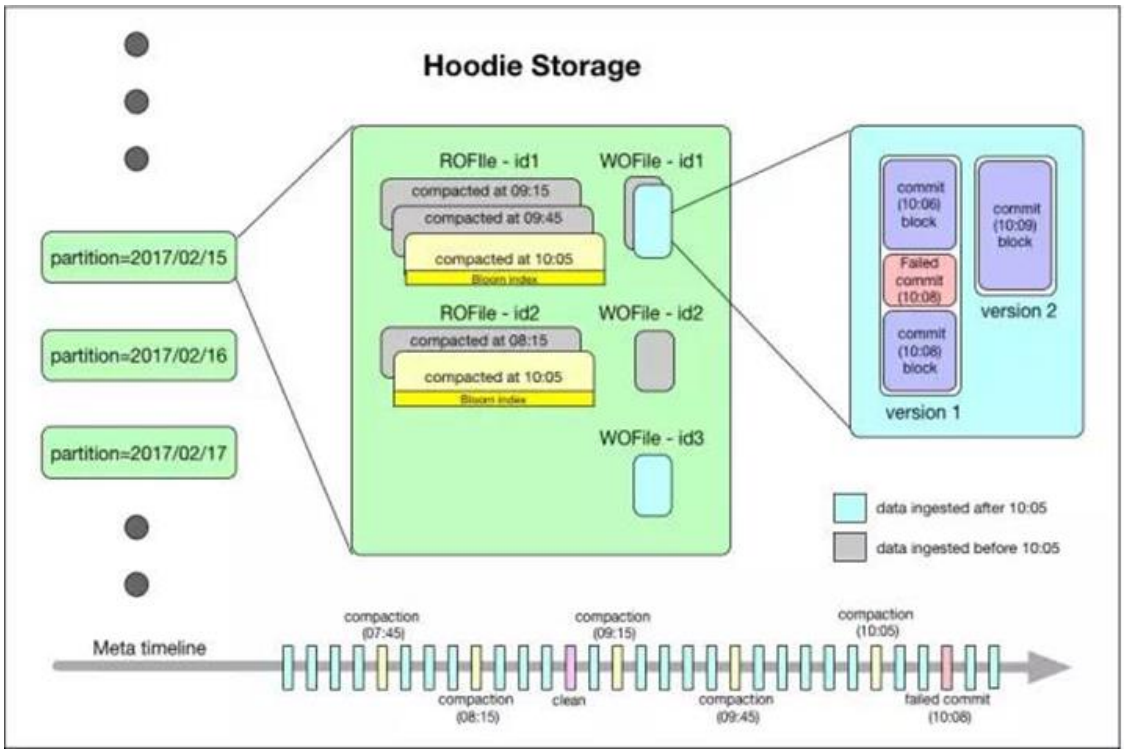
Hudi 为了实现数据的 CRUD，需要能够唯一标识一条记录，Hudi 将把数据集中的**唯一字段(record key) + 数据所在分区 (partitionPath)** 联合起来当做 **数据的唯一键**。

3.2 数据存储概述

Hudi 数据集的 **组织目录结构** 与 Hive 表示非常相似，**一份数据集对应这一个根目录**。数据集被 **打散为多个分区，分区字段以文件夹形式存在，该文件夹包含该分区的所有文件**。



在根目录下，每个分区都有唯一的分区路径，每个分区数据存储在多个文件中。



每个文件都有惟一的 fileId 和生成文件的 commit 标识。如果发生更新操作时，多个文件共享相同的 fileId，但会有不同的 commit。

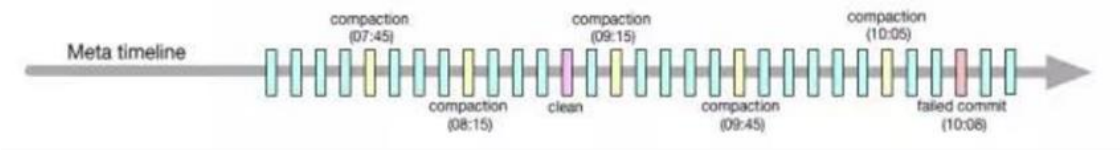
3.3 Metadata 元数据

以时间轴（**Timeline**）的形式将数据集上的各项操作元数据维护起来，以支持数据集的瞬态视图，这部分元数据存储于根目录下的元数据目录。一共有三种类型的元数据：

Commits：一个单独的 commit 包含对数据集之上一批数据的一次原子写入操作的相关信息。我们用单调递增的时间戳来标识 commits，标定的是一次写入操作的开始。

Cleans：用于清除数据集中不再被查询所用到的旧版本文件的后台活动。

Compactions：用于协调 Hudi 内部的数据结构差异的后台活动。例如，将更新操作由基于行存的日志文件归集到列存数据上

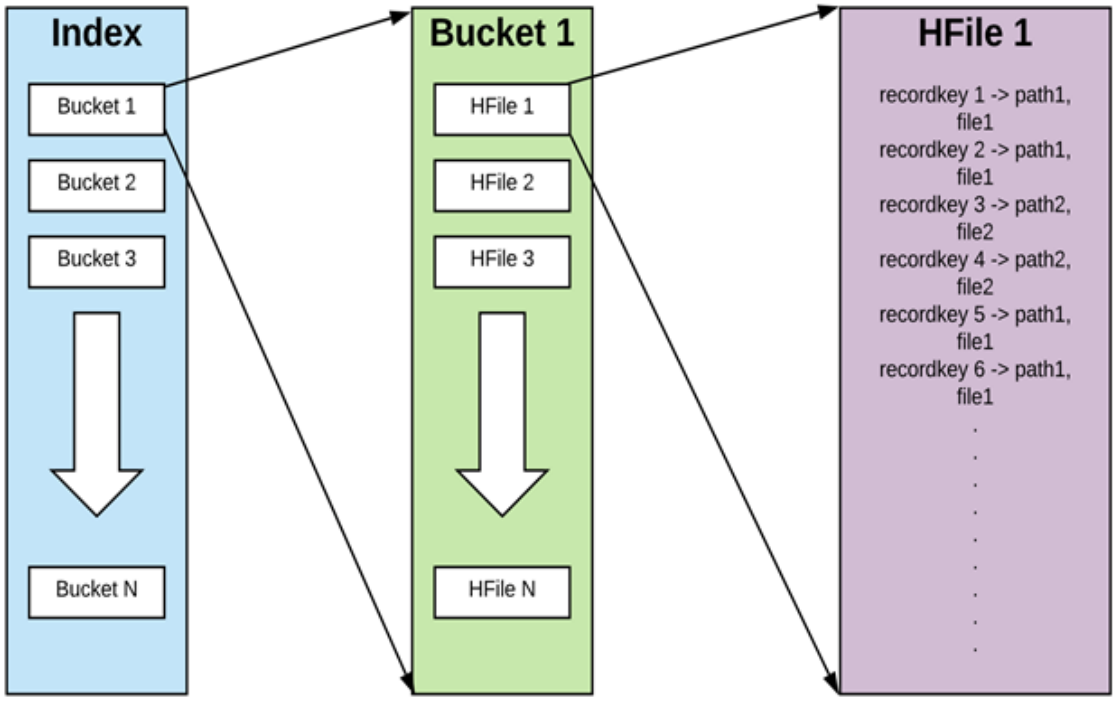


3.4 Index 索引

Hudi 维护着一个索引，以支持在记录 key 存在情况下，将新记录的 key 快速映射到对应的 fileId。

Bloom filter：存储于数据文件页脚。默认选项，不依赖外部系统实现。数据和索引始终保持一致。

Apache HBase ：可高效查找一小批 key。在索引标记期间，此选项可能快几秒钟。



3.4.1 索引策略

工作负载 1：对事实表

许多公司将大量事务数据存储在 NoSQL 数据存储中。例如，拼车情况下的行程表、股票买卖、电子商务网站中的订单。这些表通常会随着对最新数据的随机更新而不断增长，而长尾更新会针对较旧的数据，这可能是由于交易在以后结算/数据更正所致。换句话说，大多数更新进入最新的分区，很少有更新进入较旧的分区。

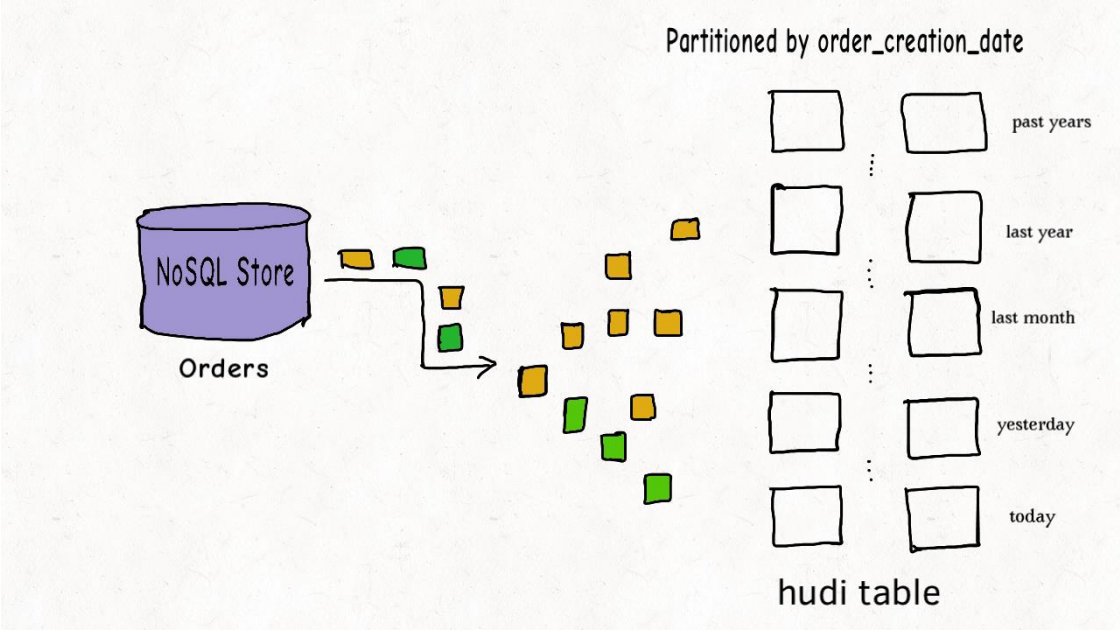


图 1：事实表的典型更新模式

对于这样的工作负载，BLOOM 索引表现良好，因为索引查找 **将基于大小合适的布隆过滤器修剪大量数据文件**。此外，如果可以构造键以使它们具有一定的顺序，则要比较的文件数量会通过范围修剪进一步减少。

Hudi 使用所有文件键范围构建一个区间树，并有效地过滤掉更新/删除记录中与任何键范围不匹配的文件。

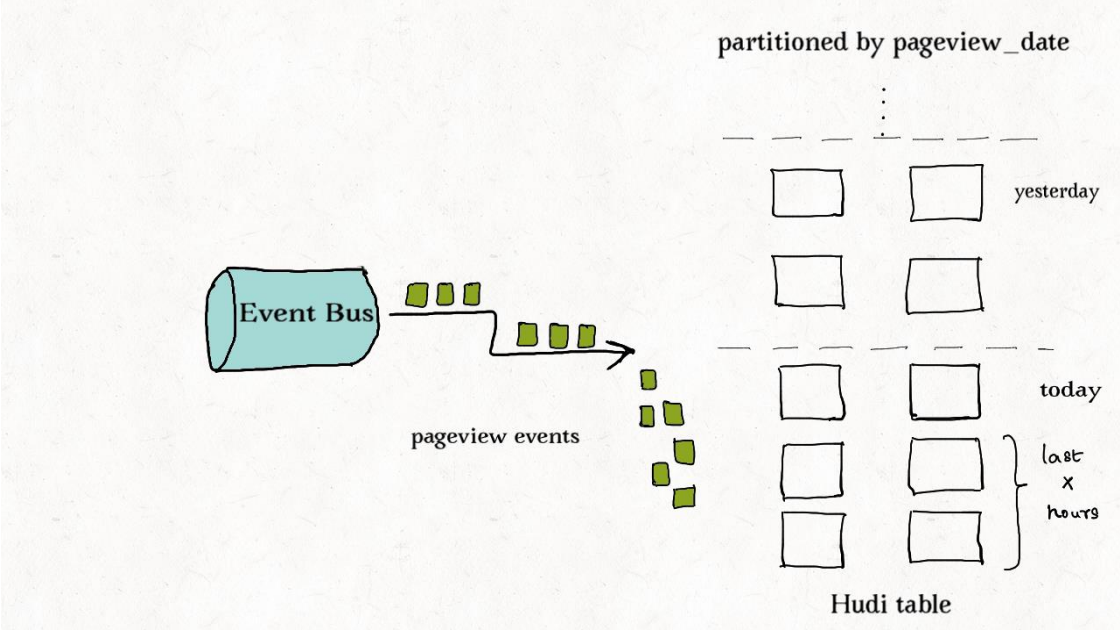
为了有效地将传入的记录键与布隆过滤器进行比较，即最小数量的布隆过滤器读取和跨执行程序的统一工作分配，Hudi 利用输入记录的缓存并采用可以使用统计信息消除数据偏差的自定义分区器。有时，如果布隆过滤器误报率很高，它可能会增加混洗的数据量以执行查找。

Hudi 支持动态布隆过滤器（使用启用 hoodie.bloom.index.filter.type=DYNAMIC_V0），它根据存储在给定文件中的记录数调整其大小，以提供配置的误报率。

工作负载 2：对事件表

事件流无处不在。来自 Apache Kafka 或类似消息总线的事件通常是事实表大小的 10-100 倍，并且通常将 **时间**（事件的到达时间/处理时间）视为一等公民。

例如，**物联网事件流、点击流数据、广告印象** 等。插入和更新仅跨越最后几个分区，因为这些大多是仅附加数据。鉴于可以在端到端管道中的任何位置引入重复事件，因此在存储到数据湖之前进行重复数据删除是一项常见要求。



一般来说，这是一个非常具有挑战性的问题，需要以较低的成本解决。虽然，我们甚至可以使用键值存储来使用 **HBASE 索引**执行重复数据删除，但索引存储成本会随着事件的数量线性增长，因此可能会非常昂贵。

实际上，BLOOM 带有范围修剪的索引是这里的最佳解决方案。人们可以利用时间通常是一等公民这一事实并构造一个键，event_ts + event_id 例如插入的记录具有单调递增的键。即使在最新的表分区中，也可以通过修剪大量文件来产生巨大的回报。

工作负载 3：随机更新/删除维度表

这些类型的表格通常包含高维数据并保存参考数据，例如**用户资料、商家信息**。这些是高保真表，其中更新通常很小，但也分布在许多分区和数据文件中，数据集从旧到新。通常，这些表也是未分区的，因为也没有对这些表进行分区的好方法。

如前所述，BLOOM 如果无法通过比较范围/过滤器来删除大量文件，则索引可能不会产生好处。在这样的随机写入工作负载中，更新最终会触及表中的大多数文件，因此布隆过滤器通常会根据一些传入的更新指示所有文件的真阳性。因此，我们最终会比较范围/过滤器，只是为了最终检查所有文件的传入更新。

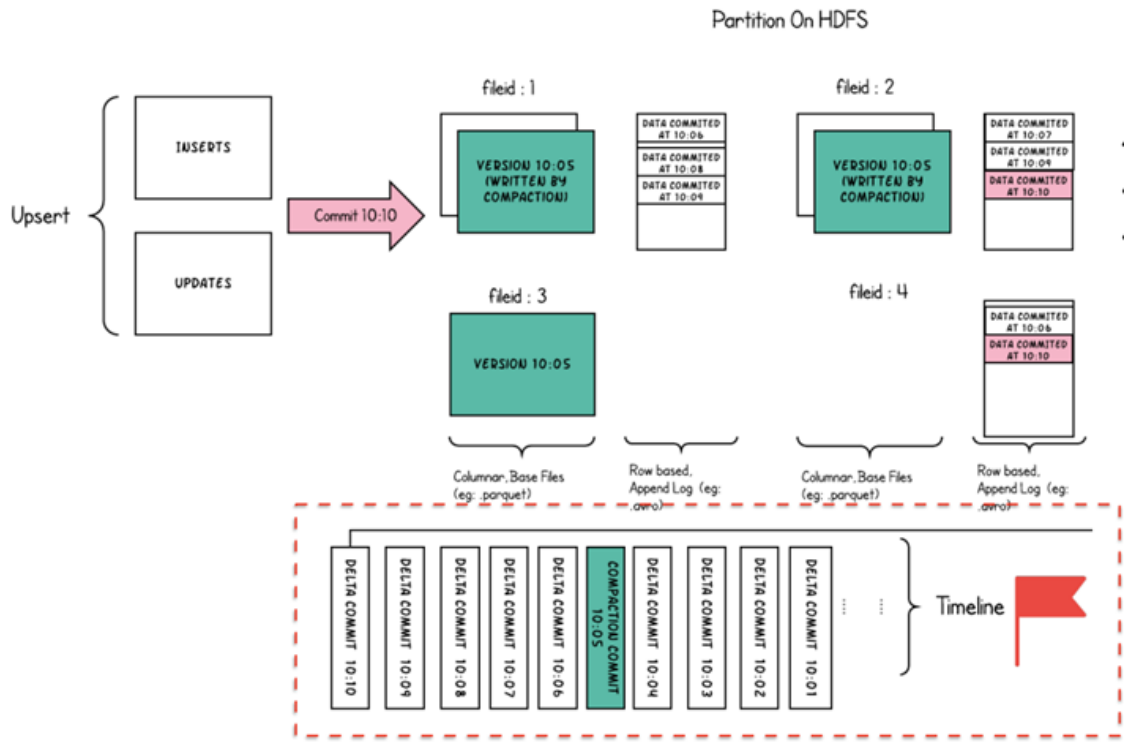
SIMPLE 索引将更适合，因为它不进行任何基于预先修剪的操作，而是直接与每个数据文件中感兴趣的字段连接 。HBASE 如果操作开销是可接受的，并且可以为这些表提供更好的查找时间，则可以使用索引。

在使用全局索引时，用户还应该考虑设置 hoodie.bloom.index.update.partition.path=true 或 hoodie.simple.index.update.partition.path=true 处理分区路径值可能因更新而改变的情况，例如用户表按家乡分区；用户搬迁到不同的城市。这些表也是 Merge-On-Read 表类型的绝佳候选者。

3.5 Data 数据

Hudi 以两种不同的存储格式存储所有摄取的数据，用户可选择满足下列条件的任意数据格式：

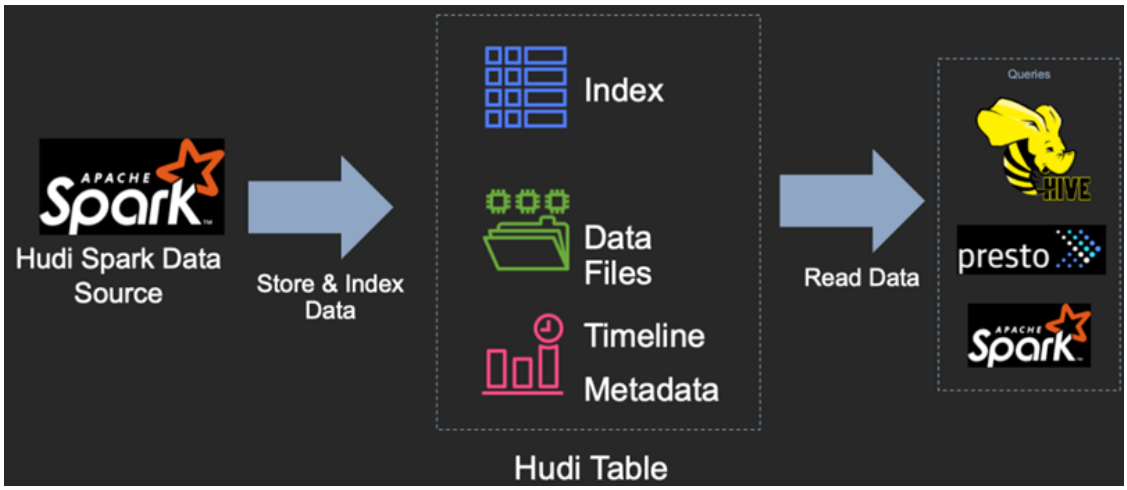
1. 读优化的列存格式（ROFormat）：缺省值为 Apache Parquet；
2. 写优化的行存格式（WOFormat）：缺省值为 Apache Avro；



4 Hudi 核心点解析

4.1 基本概念

Hudi 提供了 Hudi 表的概念，这些表支持 CRUD 操作，可以利用现有的大数据集群比如 HDFS 做数据文件存储，然后使用 SparkSQL 或 Hive 等分析引擎进行数据分析查询。



Hudi 表的三个主要组件：

1） 有序的时间轴元数据，类似于数据库事务日志。

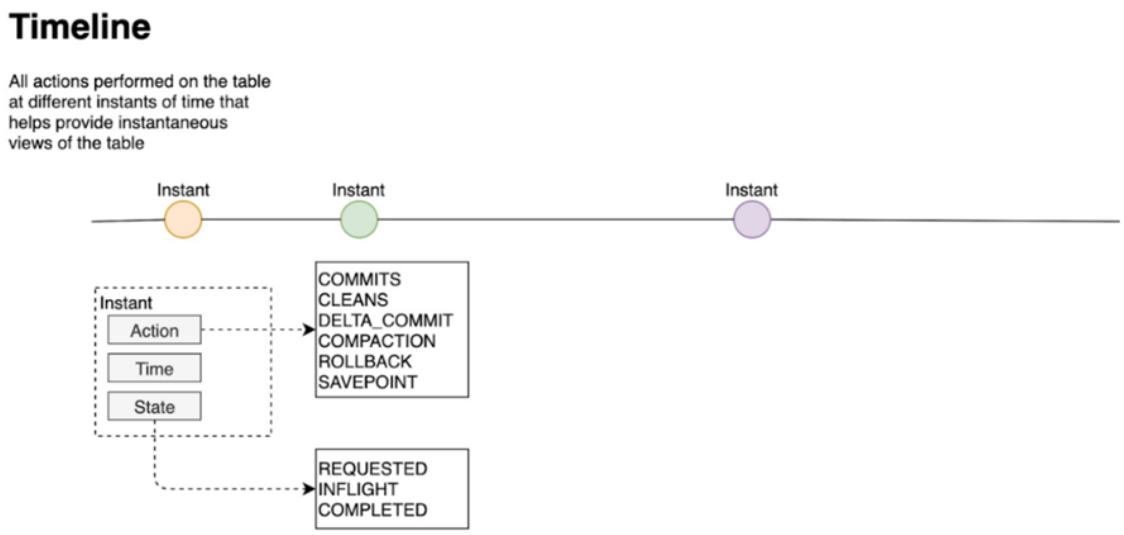
2） 分层布局的数据文件：实际写入表中的数据；

3）索引（多种实现方式）：映射包含指定记录的数据集。

4.1.1 时间轴 Timeline

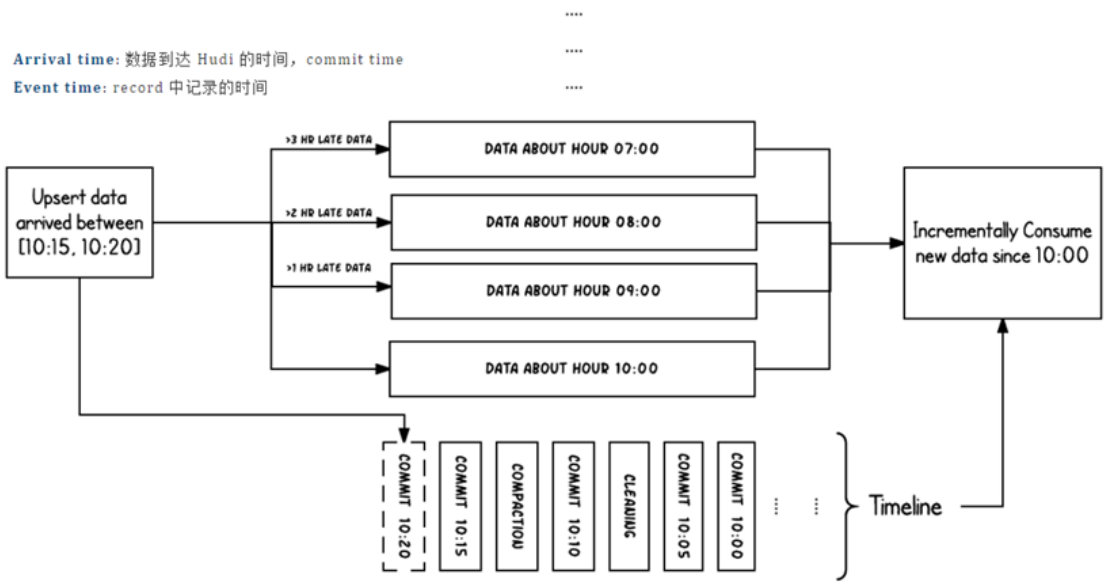
Hudi 核心：

1. 在所有的表中维护了**一个**包含**在不同的即时（Instant）时间对数据集操作**（比如新增、修改或删除）的**时间轴**（Timeline）。
2. 在**每一次对 Hudi 表的数据集操作** 时都会在**该表的 Timeline 上生成一个 Instant**，从而可以实现在仅查询某个时间点之后成功提交的数据，或是仅查询某个时间点之前的数据，有效避免了扫描更大时间范围的数据。
3. 可以高效地只查询更改前的文件（如在某个 Instant 提交了更改操作后，仅 query 某个时间点之前的数据，则仍可以 query 修改前的数据）。



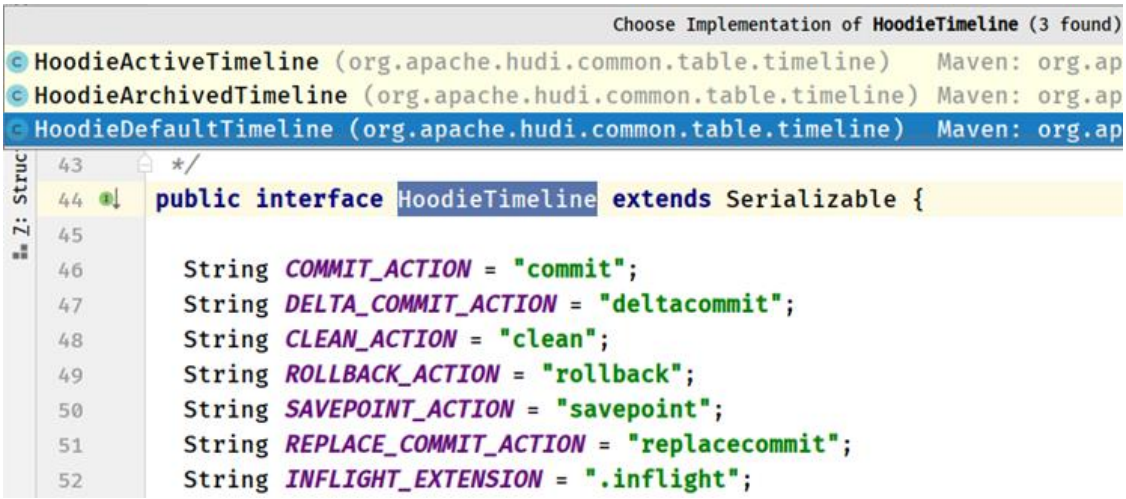
Timeline 是 Hudi 用来管理提交（commit）的抽象，每个 commit 都绑定一个固定时间戳，分散到时间线上。

在 Timeline 上，每个 commit 被抽象为一个 HoodieInstant，一个 instant 记录了一次提交 (commit) 的行为、时间戳、和状态。



图中采用时间（小时）作为分区字段，从 10:00 开始陆续产生各种 commits，10:20 来了一条 9:00 的数据，该数据仍然可以落到 9:00 对应的分区，通过 timeline 直接消费 10:00 之后的增量更新（只消费有新 commits 的 group），那么这条延迟的数据仍然可以被消费到。

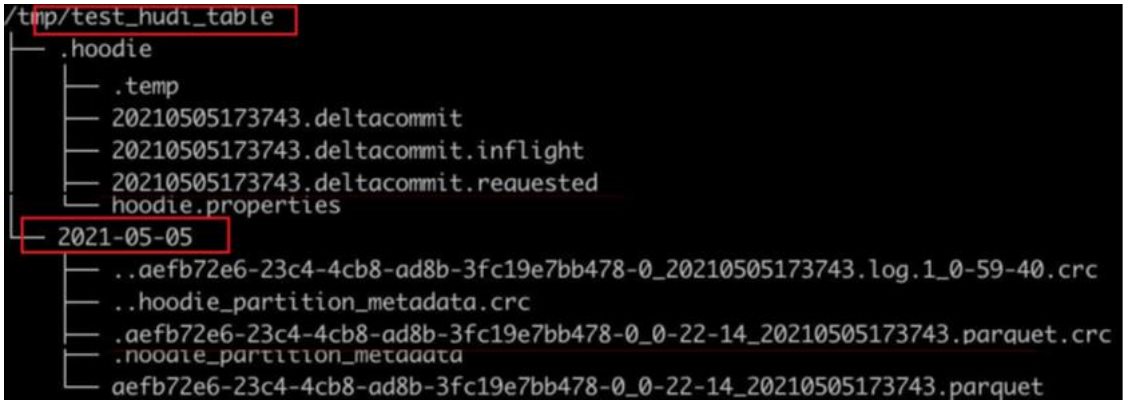
时间轴（Timeline）的实现类（位于 hudi-common-xx.jar 中）,时间轴相关的实现类位于 org.apache.hudi.common.table.timeline 包下.



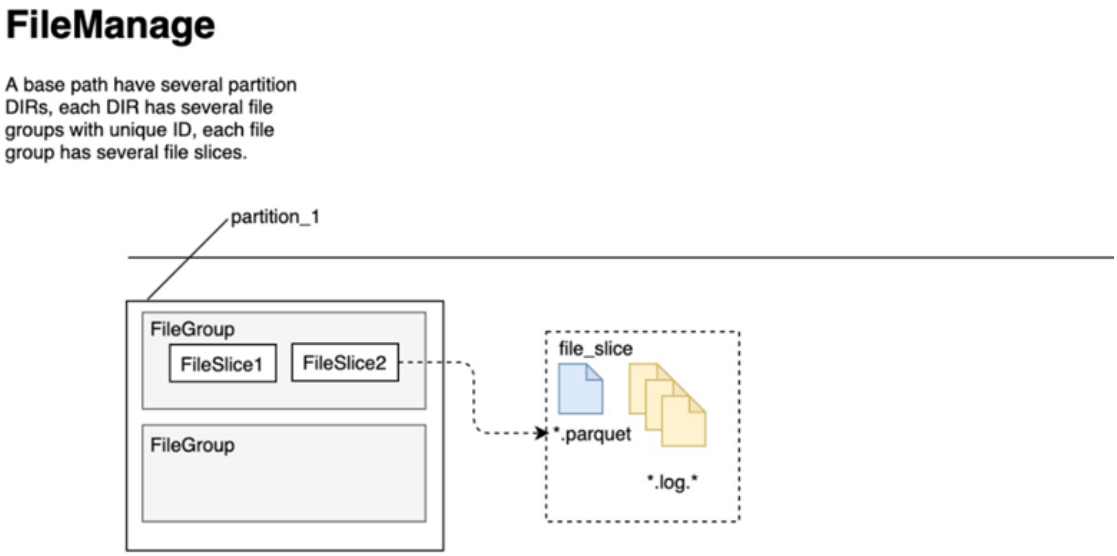
4.1.2 文件管理

Hudi 将 DFS 上的数据集组织到基本路径（HoodieWriteConfig.BASEPATHPROP）下的目录结构中。

数据集分为多个分区（DataSourceOptions.PARTITIONPATHFIELDOPT_KEY），这些分区与 Hive 表非常相似，是包含该分区的数据文件的文件夹。

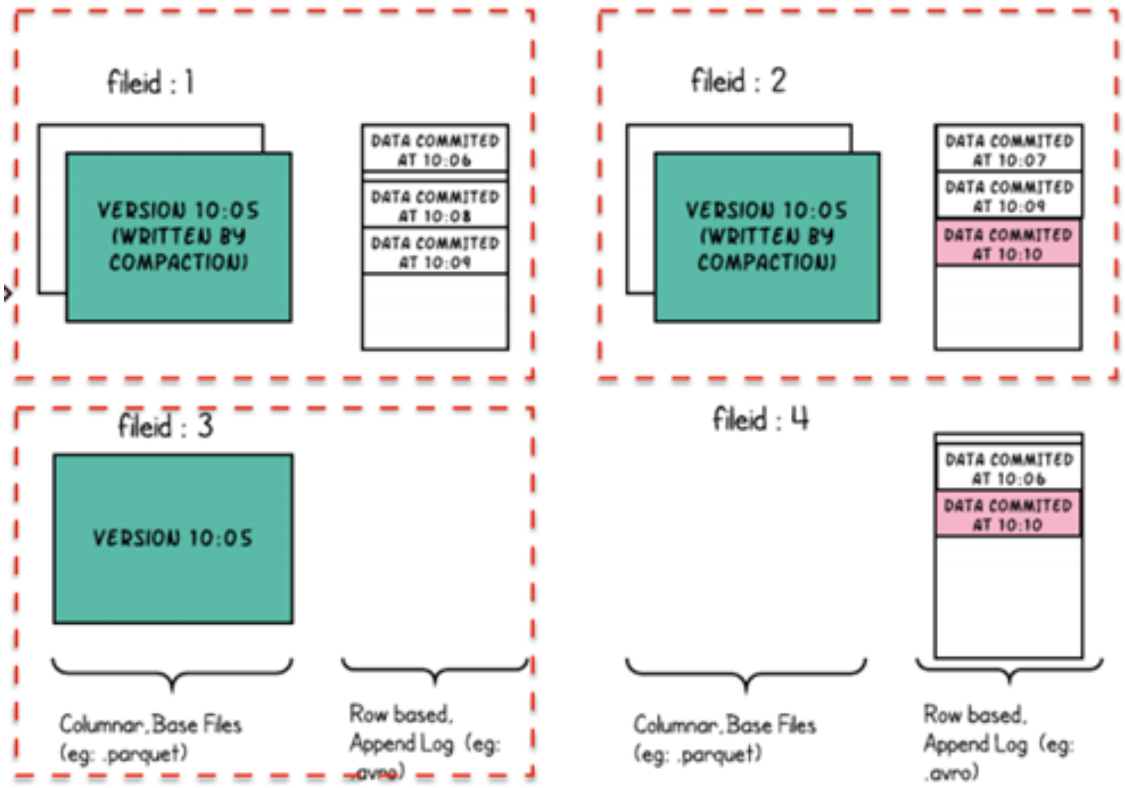


在每个分区内，文件被组织为文件组，由文件 id 充当唯一标识。每个文件组包含多个文件切片，其中每个切片包含在某个即时时间的提交/压缩生成的基本列文件（.parquet）以及一组日志文件（.log），该文件包含自生成基本文件以来对基本文件的插入/更新。



Hudi 的 base file (parquet 文件) 在 footer 的 meta 去记录了 record key 组成的 BloomFilter，用于在 file based index 的实现中实现高效率的 key contains 检测。

Hudi 的 log （avro 文件）是自己编码的，通过积攒数据 buffer 以 LogBlock 为单位写出，每个 LogBlock 包含 magic number、size、content、footer 等信息，用于数据读、校验和过滤。



4.1.3 索引 Index

- Hudi 通过**索引**机制提供**高效的 Upsert 操作**，该机制会将一个 **RecordKey+PartitionPath** 组合的方式作为唯一标识映射到一个文件 ID，而且这个唯一标识和文件组/文件 ID 之间的映射自记录被写入文件组开始就不会再改变。

- **全局索引**：在全表的所有分区范围下强制要求键保持唯一，即确保对给定的键有且只有一个对应的记录。

- **非全局索引**：仅在表的某一个分区内强制要求键保持唯一，它依靠写入器为同一个记录的更删提供一致的分区路径。

```
scala> tripsSnapshotDF.printSchema()
root
 |-- _hoodie_commit_time: string (nullable = true)
 |-- _hoodie_commit_seqno: string (nullable = true)
 |-- _hoodie_record_key: string (nullable = true)
 |-- _hoodie_partition_path: string (nullable = true)
 |-- _hoodie_file_name: string (nullable = true)
 |-- begin_lat: double (nullable = true)
 |-- begin_lon: double (nullable = true)
 |-- driver: string (nullable = true)
 |-- end_lat: double (nullable = true)
 |-- end_lon: double (nullable = true)
 |-- fare: double (nullable = true)
 |-- partitionpath: string (nullable = true)
 |-- rider: string (nullable = true)
 |-- ts: long (nullable = true)
 |-- uuid: string (nullable = true)
```

4.2 表的存储类型

4.2.1 数据计算模型

- Hudi 是 Uber 主导开发的开源数据湖框架，所以大部分的出发点都来源于 Uber 自身场景，比如司机数据和乘客数据通过订单 Id 来做 Join 等。
- 在 Hudi 过去的使用场景里，和大部分公司的架构类似，采用批式和流式共存的 Lambda 架构，后来 Uber 提出增量 Incremental 模型，相对批式来讲，更加实时；相对流式而言，更加经济。

4.2.1.1 批式模型（Batch）

批式模型就是使用 MapReduce、Hive、Spark 等典型的批计算引擎，**以小时任务或者天任务的形式来做数据计算**。

1. 延迟：小时级延迟或者天级别延迟。这里的延迟不单单指的是定时任务的时间，在数据架构里，这里的延迟时间通常是定时任务间隔时间 + 一系列依赖任务的计算时间 + 数据平台最终可以展示结果的时间。数据量大、逻辑复杂的情况下，小时任务计算的数据通常真正延迟的时间是 2-3 小时。

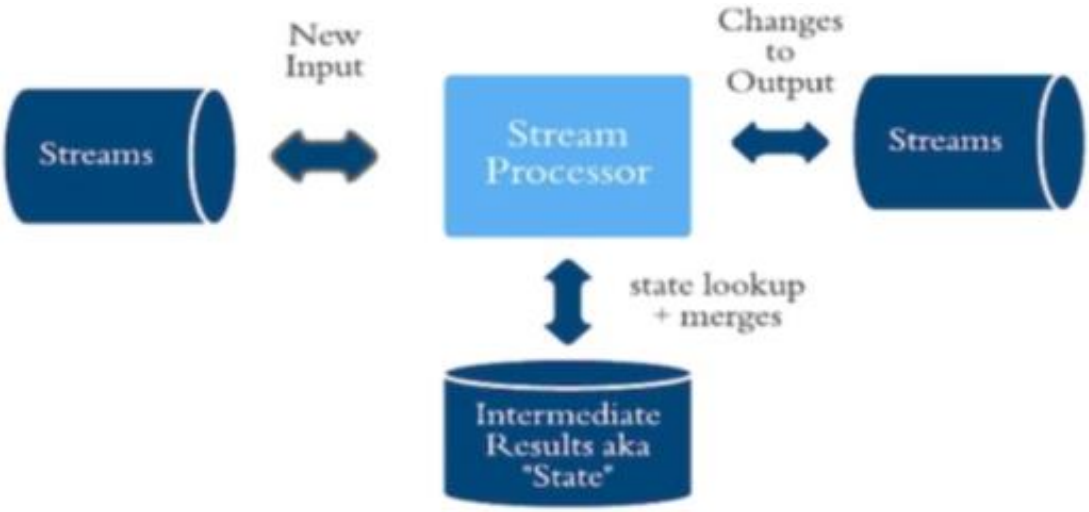
2. 数据完整度：数据较完整。以处理时间为例，小时级别的任务，通常计算的原始数据已经包含了小时内的所有数据，所以得到的数据相对较完整。但如果业务需求是事件时间，这里涉及到终端的一些延迟上报机制，在这里，批式计算任务就很难派上用场。

3. 成本：成本很低。只有在做任务计算时，才会占用资源，如果不做任务计算，可以将这部分批式计算资源出让给在线业务使用。从另一个角度来说成本是挺高的，如原始数据做了一些增删改查，数据晚到的情况，那么批式任务是要全量重新计算。

4.2.1.2 流式模型（Stream）

流式模型，典型的就是使用 Flink 来进行**实时的数据计算**。

1. 延迟：很短，甚至是实时。
2. 数据完整度：较差。因为流式引擎不会等到所有数据到齐之后再开始计算，所以有一个 watermark 的概念，当数据的时间小于 watermark 时，就会被丢弃，这样是无法对数据完整度有一个绝对的报障。在互联网场景中，流式模型主要用于活动时的数据大盘展示，对数据的完整度要求并不算很高。在大部分场景中，用户需要开发两个程序，一是流式数据生产流式结果，二是批式计算任务，用于次日修复实时结果。
3. 成本：很高。因为流式任务是常驻的，并且对于多流 Join 的场景，通常要借助内存或者数据库来做 state 的存储，不管是序列化开销，还是和外部组件交互产生的额外 IO，在大数据量下都是不容忽视的。



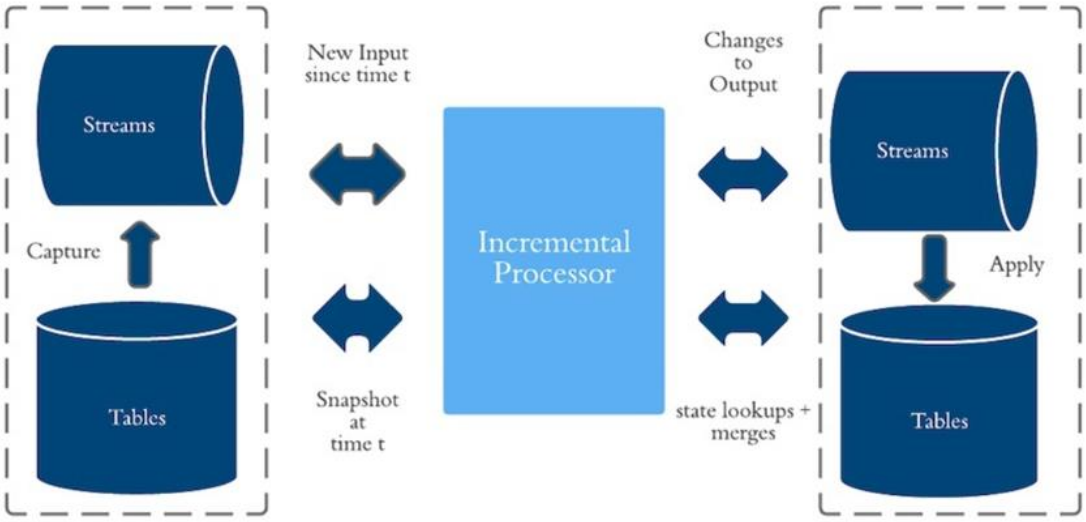
4.2.1.3 增量模型（Incremental）

针对批式和流式的优缺点，Uber 提出了增量模型（Incremental Mode），相对批式来讲，更加实时；相对流式而言，更加经济。

增量模型，简单来讲，是以 mini batch 的形式来跑准实时任务。Hudi 在增量模型中支持了两个最重要的特性：

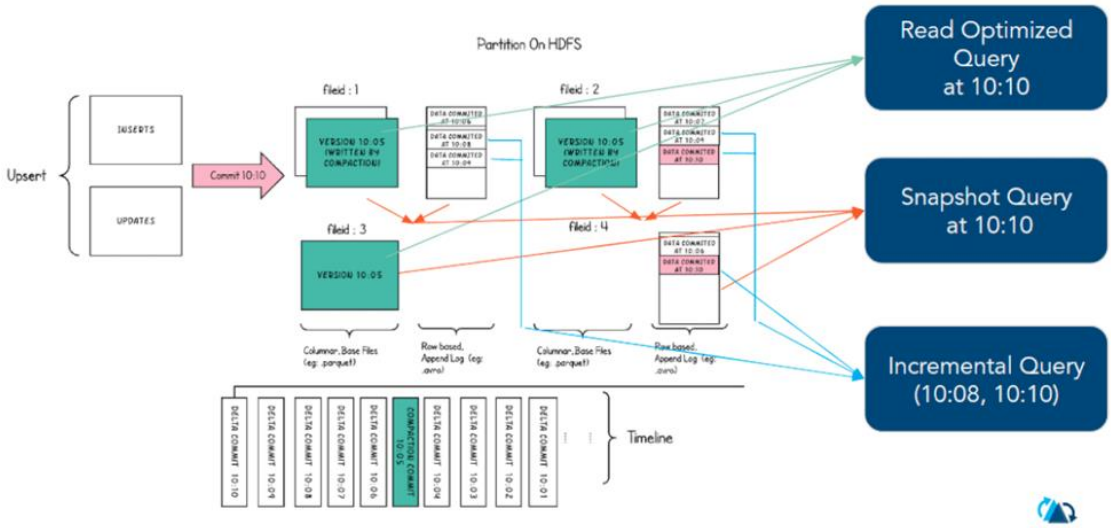
1. Upsert：这个主要是解决批式模型中，数据不能插入、更新的问题，有了这个特性，可以往 Hive 中写入增量数据，而不是每次进行完全的覆盖。（Hudi 自身维护了 key->file 的映射，所以当 upsert 时很容易找到 key 对应的文件）

2. Incremental Query：增量查询，减少计算的原始数据量。以 Uber 中司机和乘客的数据流 Join 为例，每次抓取两条数据流中的增量数据进行批式的 Join 即可，相比流式数据而言，成本要降低几个数量级。



4.2.2 查询类型（Query Type）

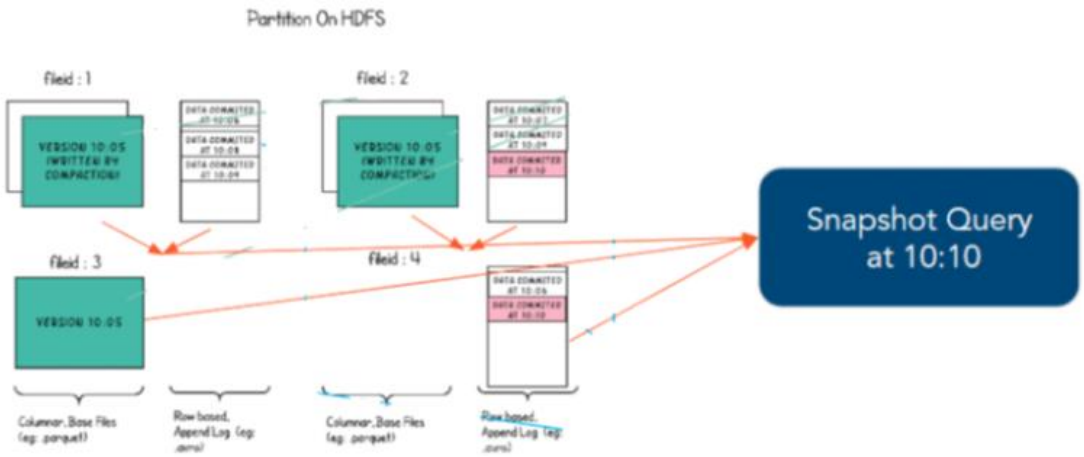
Hudi 支持三种不同的查询表的方式：Snapshot Queries、Incremental Queries 和 Read Optimized Queries。



4.2.2.1 快照查询（Snapshot Queries）

类型一：**Snapshot Queries（快照查询）**

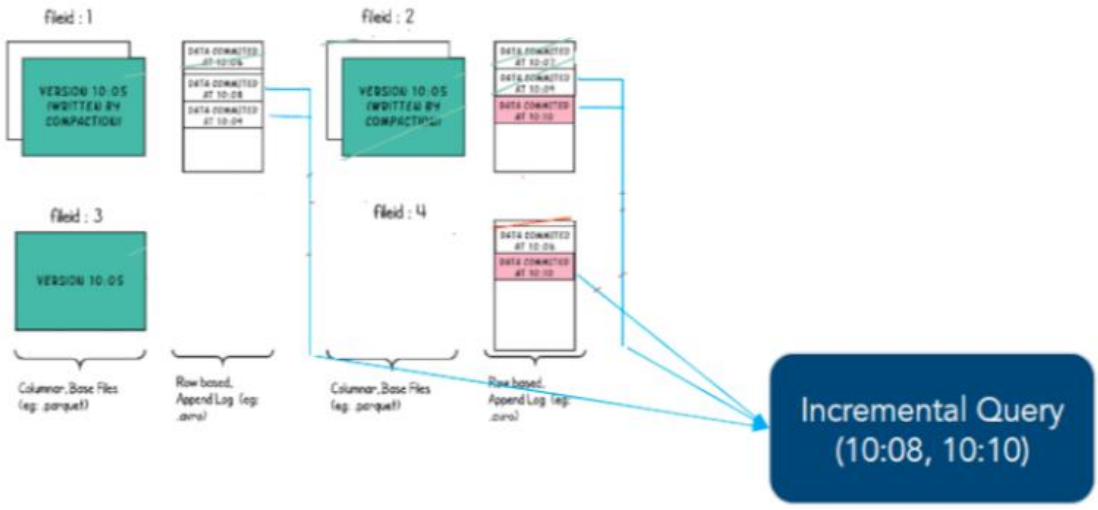
- 查询某个增量提交操作中数据集的最新快照，先进行动态合并最新的基本文件(Parquet)和增量文件(Avro)来提供近实时数据集（通常会存在几分钟的延迟）。
- 读取所有 partiiton 下每个 FileGroup 最新的 FileSlice 中的文件，Copy On Write 表读 parquet 文件，Merge On Read 表读 parquet + log 文件



4.2.2.2 增量查询（Incremental Queries）

类型二：**Incremental Queries（增量查询）**

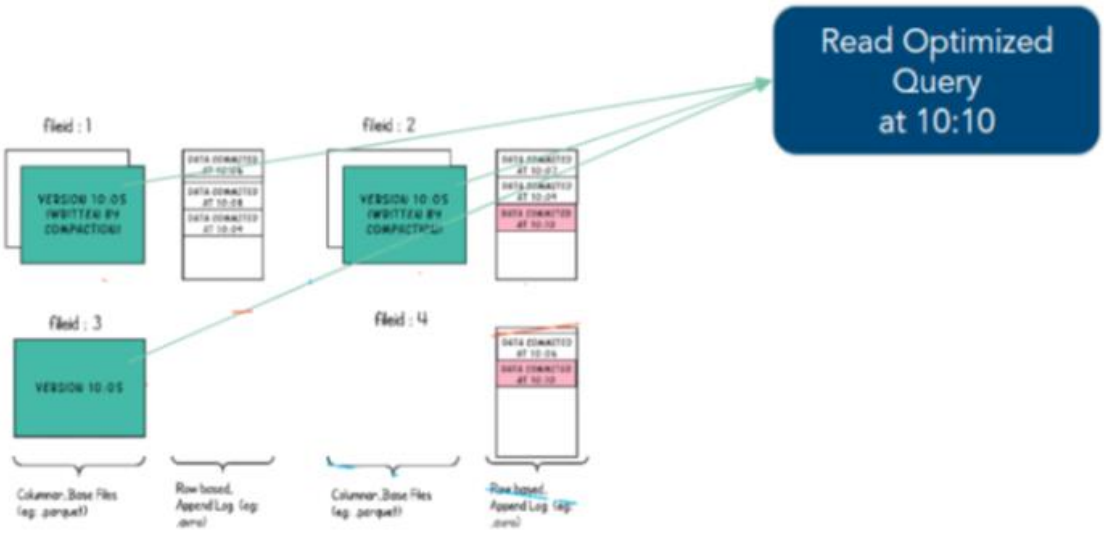
- 仅查询新写入数据集的文件，需要指定一个 Commit/Compaction 的即时时间（位于 Timeline 上的某个 Instant）作为条件，来查询此条件之后的新数据。
- 可查看自给定 commit/delta commit 即时操作以来新写入的数据，有效的提供变更流来启用增量数据管道。



4.2.2.3 读优化查询（Read Optimized Queries）

类型三：**Read Optimized Queries（读优化查询）**

- 直接查询基本文件（数据集的最新快照），其实就是列式文件（Parquet）。并保证与非 Hudi 列式数据集相比，具有相同的列式查询性能。
- 可查看给定的 commit/compact 即时操作的表的最新快照。
- 读优化查询和快照查询相同仅访问基本文件，提供给定文件片自上次执行压缩操作以来的数据。通常查询数据的最新程度的保证取决于压缩策略



4.2.3 Hudi 支持表类型

Hudi 提供两类型表：**写时复制（Copy on Write，COW）表和读时合并（Merge On Read，MOR）表**。

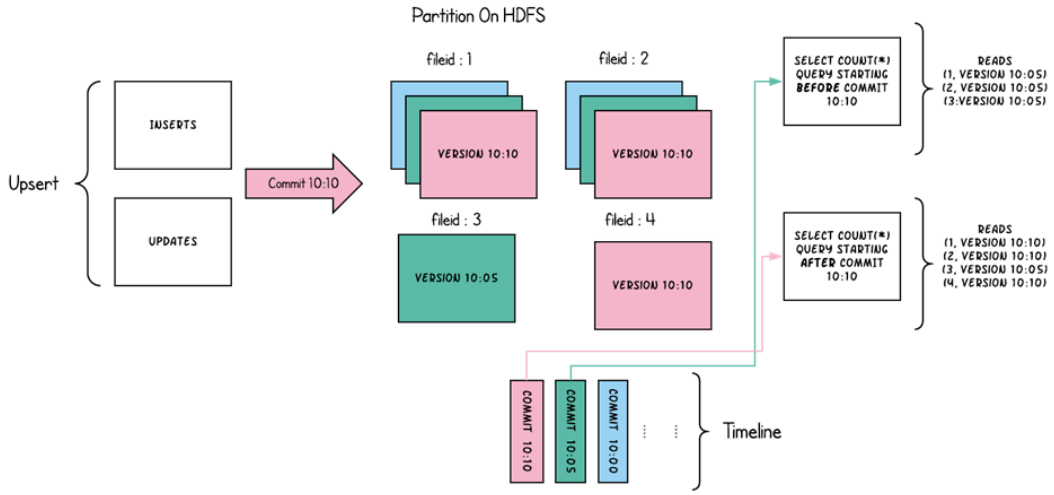
- 对于 Copy-On-Write Table，用户的 update 会重写数据所在的文件，所以是一个写放大很高，但是读放大为 0，适合写少读多的场景。
- 对于 Merge-On-Read Table，整体的结构有点像 LSM-Tree，用户的写入先写入到 delta data 中，这部分数据使用行存，这部分 delta data 可以手动 merge 到存量文件中，整理为 parquet 的列存结构。

Table Type	Supported Query types
Copy On Write	Snapshot Queries + Incremental Queries
Merge On Read	Snapshot Queries + Incremental Queries + Read Optimized Queries

4.2.3.1 写时复制表（COW）

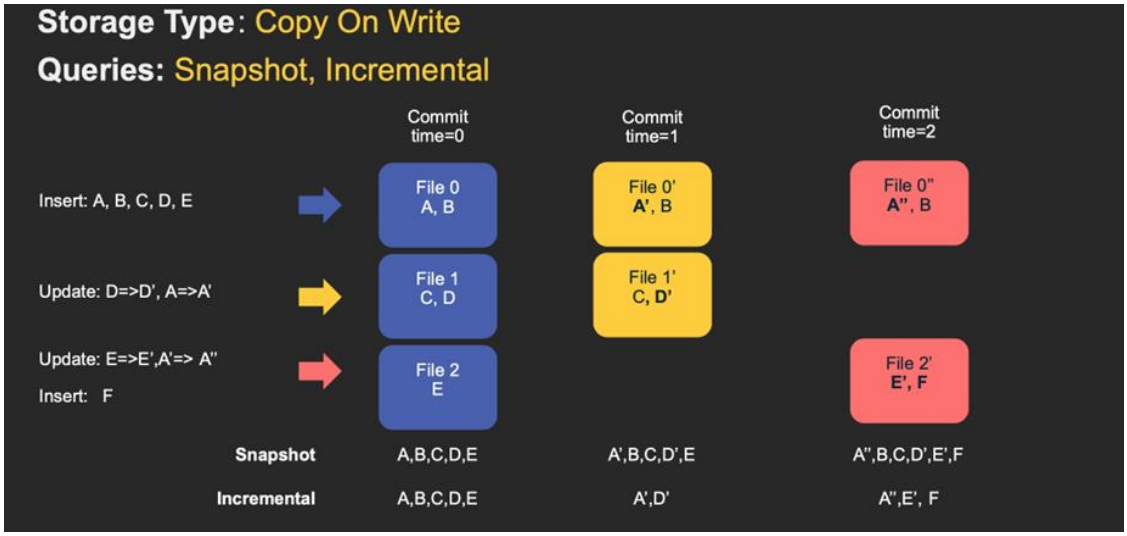
Copy on Write 简称 COW，顾名思义，它是**在数据写入的时候，复制一份原来的拷贝，在其基础上添加新数据**。

正在读数据的请求，读取的是最近的完整副本，这类似 Mysql 的 MVCC 的思想。



- 优点：读取时，只读取对应分区的一个数据文件即可，较为高效；

- 缺点：数据写入的时候，需要复制一个先前的副本再在其基础上生成新的数据文件，这个过程比较耗时



COW 表主要**使用列式文件格式（Parquet）存储数据，在写入数据过程中，执行同步合并，更新数据版本并重写数据文件**，类似 RDBMS 中的 B-Tree 更新。

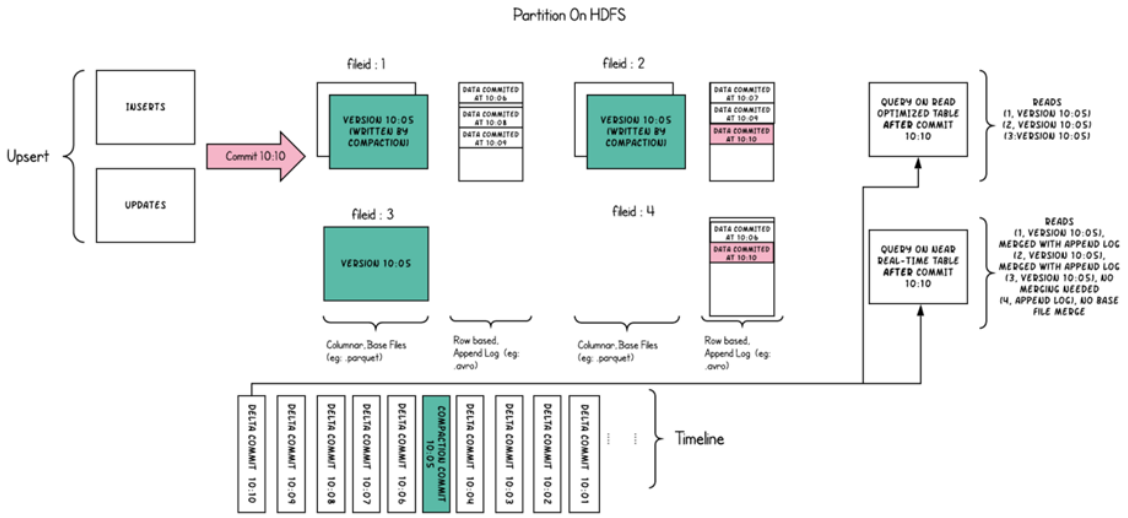
1. 更新 update：在更新记录时，Hudi 会先找到包含更新数据的文件，然后再使用更新值（最新的数据）重写该文件，包含其他记录的文件保持不变。当突然有大量写操作时会导致重写大量文件，从而导致极大的 I/O 开销。

2. 读取 read：在读取数据时，通过读取最新的数据文件来获取最新的更新，此存储类型适用于少量写入和大量读取的场景

4.2.3.2 读时合并表（MOR）

Merge On Read 简称 MOR，**新插入的数据存储在 delta log 中，定期再将 delta log 合并进行 parquet 数据文件**。

读取数据时，会将 delta log 跟老的数据文件做 merge，得到完整的数据返回。下图演示了 MOR 的两种数据读写方式



- 优点：由于写入数据先写 delta log，且 delta log 较小，所以写入成本较低；
- 缺点：需要定期合并整理 compact，否则碎片文件较多。读取性能较差，因为需要将 delta log 和老数据文件合并；

MOR 表是 COW 表的升级版，它使用列式（parquet）与行式（avro）文件混合的方式存储数据。在更新记录时，类似 NoSQL 中的 LSM-Tree 更新。

1. 更新：在更新记录时，仅更新到增量文件（Avro）中，然后进行异步（或同步）的 compaction，最后创建列式文件（parquet）的新版本。此存储类型适合频繁写的工作负载，因为新记录是以追加的模式写入增量文件中。
2. 读取：在读取数据集时，需要先将增量文件与旧文件进行合并，然后生成列式文件成功后，再进行查询。

4.2.3.3 COW VS MOR

对于写时复制（COW）和读时合并（MOR）writer 来说，Hudi 的 WriteClient 是相同的。

- COW 表，用户在 snapshot 读取的时候会扫描所有最新的 FileSlice 下的 base file。
- MOR 表，在 READ OPTIMIZED 模式下，只会读最近的经过 compaction 的 commit。

权衡	写时复制COW	读时合并MOR
数据延迟	更高	更低
更新代价(I/O)	更高（重写整个parquet文件）	更低（追加到增量日志）
Parquet文件大小	更小（高更新代价（I/O））	更大（低更新代价）
写放大	更高	更低（取决于压缩策略）
适用场景	写少读多	写多读少

4.2.4 数据写操作类型

在 Hudi 数据湖框架中支持三种方式写入数据：**UPSERT（插入更新）、INSERT（插入）和 BULK INSERT（写排序）**。

- **UPSERT**：默认行为，数据先通过 index 打标(INSERT/UPDATE)，有一些启发式算法决定消息的组织以优化文件的大小
- **INSERT**：跳过 index，写入效率更高
- **BULK**_INSERT**：写排序，对大数据量的 Hudi 表初始化友好，对文件大小的限制 best effort（写 HFile）

4.2.4.1 写流程(upsert)

（1）Copy On Write 类型表，UPSERT 写入流程

第一步、先对 records 按照 record key 去重；

第二步、首先对这批数据创建索引 (HoodieKey => HoodieRecordLocation)；通过索引区分哪些 records 是 update，哪些 records 是 insert（key 第一次写入）；

第三步、对于 update 消息，会直接找到对应 key 所在的最新 FileSlice 的 base 文件，并做 merge 后写新的 base file (新的 FileSlice)；

第四步、对于 insert 消息，会扫描当前 partition 的所有 SmallFile（小于一定大小的 base file），然后 merge 写新的 FileSlice；如果没有 SmallFile，直接写新的 FileGroup + FileSlice；

（2）Merge On Read 类型表，UPSERT 写入流程

第一步、先对 records 按照 record key 去重（可选）

第二步、首先对这批数据创建索引 (HoodieKey => HoodieRecordLocation)；通过索引区分哪些 records 是 update，哪些 records 是 insert（key 第一次写入）

第三步、如果是 insert 消息，如果 log file 不可建索引（默认），会尝试 merge 分区内最小的 base file （不包含 log file 的 FileSlice），生成新的 FileSlice；如果没有 base file 就新写一个 FileGroup + FileSlice + base file；如果 log file 可建索引，尝试 append 小的 log file，如果没有就新写一个 FileGroup + FileSlice + base file

第四步、如果是 update 消息，写对应的 file group + file slice，直接 append 最新的 log file（如果碰巧是当前最小的小文件，会 merge base file，生成新的 file slice）log file 大小达到阈值会 roll over 一个新的

4.2.4.2 写流程(Insert)

(1) Copy On Write 类型表，INSERT 写入流程

第一步、先对 records 按照 record key 去重（可选）；

第二步、不会创建 Index；

第三步、如果有小的 base file 文件，merge base file，生成新的 FileSlice + base file，否则直接写新的 FileSlice + base file；

(2) Merge On Read 类型表，INSERT 写入流程

第一步、先对 records 按照 record key 去重（可选）；

第二步、不会创建 Index；

第三步、如果 log file 可索引，并且有小的 FileSlice，尝试追加或写最新的 log file；如果 log file 不可索引，写一个新的 FileSlice + base file；