# 背景

摘自官网：Apache Iceberg is an open table format for huge analytic datasets.，可以看到 Founders 对 Iceberg 的定位是面向海量数据分析场景的高效存储格式。海量数据分析的场景，类比于 Hive 是 Hdfs 的封装一样，本质上解决的还是数仓场景的痛点问题。

Iceberg 在最开始，也确实是在数仓场景朝着更快更好用的 format 目标不断演进，比如支持 schema 变更，文件粒度的 Filter 优化等，但随着和流式计算 Flink 引擎的生态打通，Delete/Update/Merge 语义的出现，场景就会变得多样化起来。

过去业界更多是使用 Hive/Spark on HDFS 作为离线数据仓库的载体，在越来越趋于实时化和快速迭代的场景中，逐渐暴露出以下缺点：

不支持 Row-Level-Update，对于更新的操作需要 overwrite 整张 Hive 表，成本极高

不支持读写分离，用户的读取操作会被另一个用户的写入操作所影响（尤其是流式读取的场景）

不支持版本回滚和快照，需要保存大量历史数据

不支持增量读取，每次扫描全表或分区所有数据

性能低，只能裁剪到 Hive Partition 粒度

不支持 Schema 变更

…..

# 概述

## 简介

## Hudi VS Iceberg



选择 Hudi 的场景：

1、需要频繁的增量数据写入（如日志数据、IoT 数据等）。

2、对数据更新（upsert）和删除（delete）操作有较高要求。

3、需要流处理或增量处理能力（Flink、Kafka 等集成）。

选择 Iceberg 的场景：

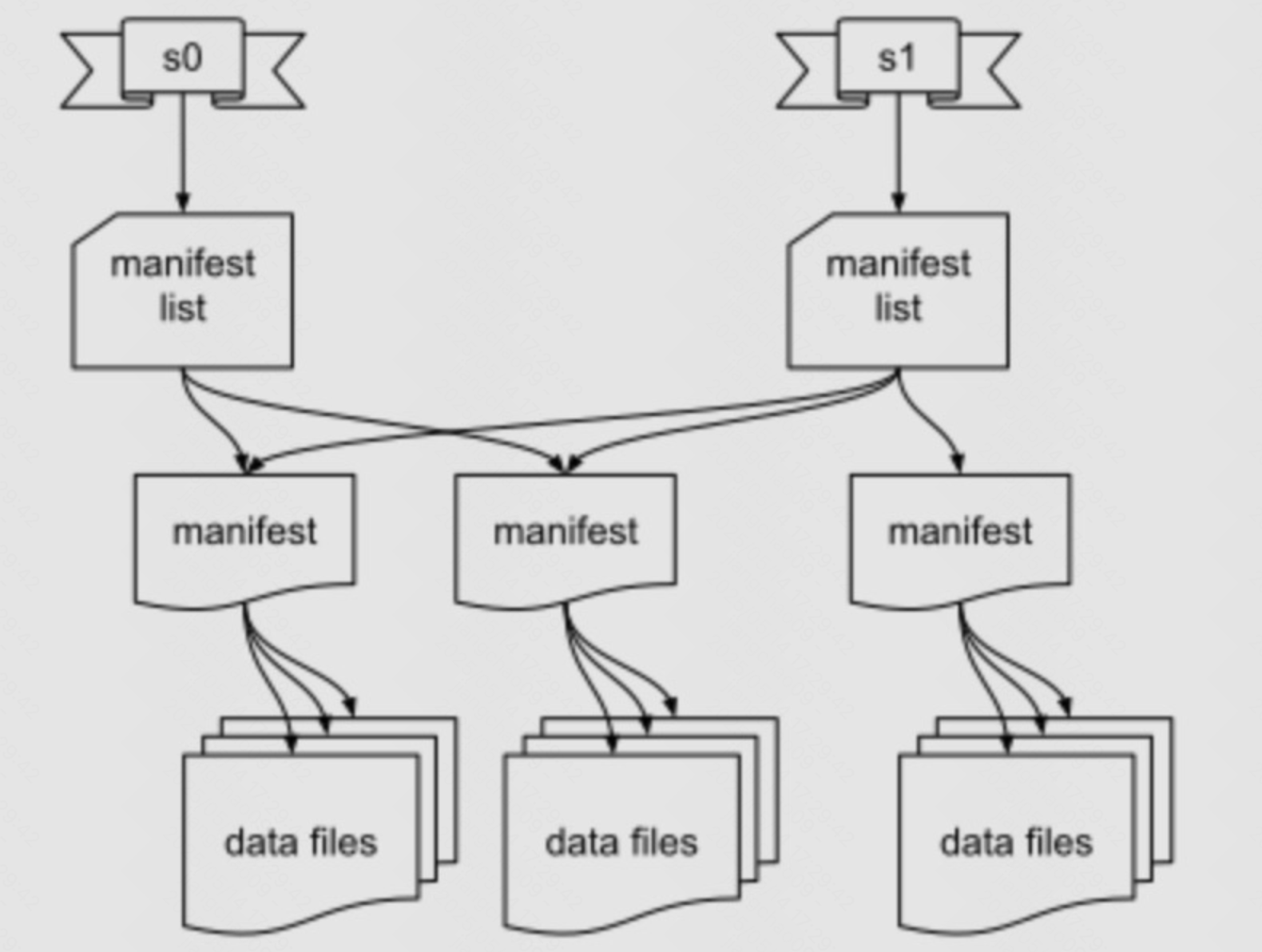
1、需要高效的、大规模的数据表管理和查询（如数据仓库分析、批量计算等）。

2、需要支持时间旅行（回溯历史数据）和多版本并发控制（MVCC）。

3、需要灵活的分区管理和动态优化查询。

## 基本概念

如上图所示，iceberg 将 hdfs 上的文件进行了 snapshot、manifest list、manifest、data files 的分层。



Snapshot：用户的每次 commit（每次写入的 spark job） 会产生一个新的 snapshot

Manifest List：维护当前 snapshot 中所有的 manifest

Manifest：维护当前 Manifest 下所有的 data files

Data File：存储数据的文件，后续 Iceberg 引入了 Delete File，用于存储要删除的数据，文件结构上也是与 Data File 处在同一层

# 核心功能

## Time Travel 和增量读取

Time Travel 指的是用户可以任意读取历史时刻的相关数据，以 Spark 的代码为例：

// time travel to October 26, 1986 at 01:21:00

spark.read

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

.format("iceberg")

.load("path/to/table")

上述代码即是在读取 timestamp=499162860000 时，该 Iceberg 表的数据，那么底层原理是什么样子的呢？

从「基本概念」中的文件结构可以看到，用户每次新的写入都会产生一个 snapshot，那么 Iceberg 只需要存储用户每次 commit 产生的 metadata，比如时间戳等信息，就能找到对应时刻的 snapshot，并且解析出 Data Files。

增量读取也同理，通过 start 和 end 的时间戳取到时间范围内的 snapshot，并读取所有的 Data Files 作为原始数据。

## Fast Scan & Data Filtering

上面提到 Hive 的查询性能低下，其中一个原因是数据计算时，只能下推到 Partition 层面，粒度太粗。而 Iceberg 在细粒度的 Plan 上做了一系列的优化，当一个 Query 进入 Iceberg 后：

1、根据 timestamp 找到对应的 snapshot（默认最新）

2、根据 Query 的 Partition 信息从指定 snapshot 中过滤出符合条件的 manifest 文件集合

3、从 manifest 文件集合中取出所有的 Data Files 对象（只包含元信息）

4、根据 Data File 的若干个属性，进行更细粒度的数据过滤，包括 column-level value counts, null counts, lower bounds, and upper bounds 等

## Delete 实现

为了上线 Row-Level Update 的功能，Iceberg 提供了 Delete 的实现，通过 Delete + Insert 我们可以达到 Update 的目的。在引入 Delete 实现时，引入了两个概念：

Delete File：用于存储删除的数据（分为 position delete 和 equality delete）

Sequence Number：是 Data File 和 Delete File 的共有属性之一，主要用于区分 Insert 和 Delete 的先后顺序，否则会出现数据一致性的问题

**position & equality delete**

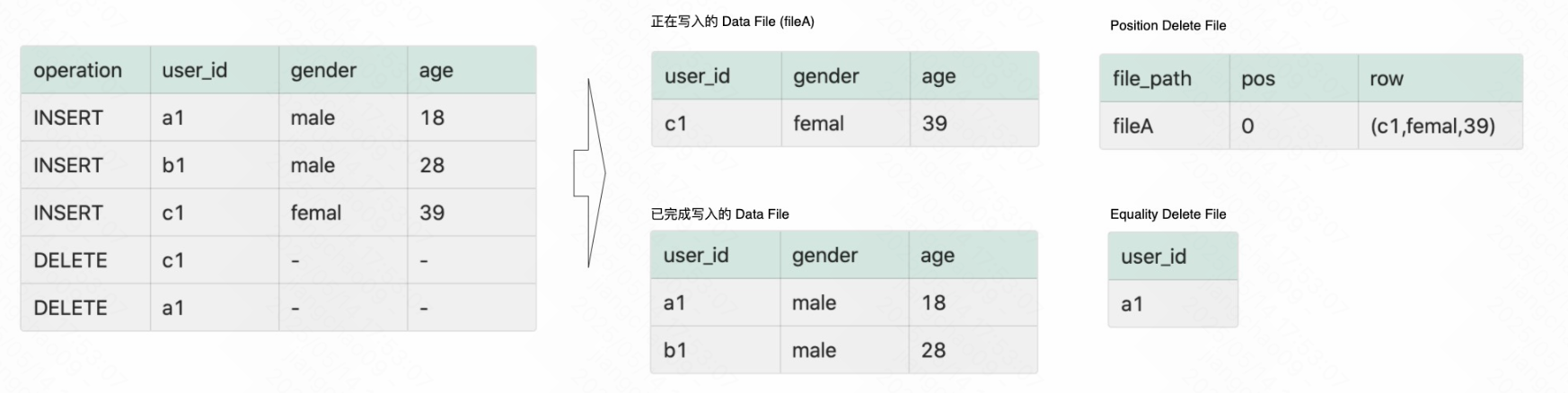
Iceberg 引入了 equality\_ids 概念，用户建表时可以指定 Table 的 equality\_ids 来标识未来 Delete 操作对应的 Key，比如 GDPR 场景，我们需要根据 user\_id 来随机删除用户的相关数据，就可以把 equality\_ids 设置为 user\_id。

两种 Delete 操作对应不同的 Delete File，其存储字段也不同：

position delete：包括三列，file\_path（要删除的数据所在的 Data File）、pos（行数）、row（数据）

equality delete：包括 equality\_ids 中的字段

显而易见，存储 Delete File 的目的是将来读取数据时，进行实时的 Join，而 position delete 在 Join 时能精准定位到文件，并且只需要行号的比较，肯定是更加高效的。所以在 Delete 操作写入时，Iceberg 会将正在写入的数据文件信息存储到内存中，来保证将 DELETE 操作尽量走 position delete 的链路。示意图如下所示：



按照时间顺序，依次写入三条 INSERT 和 DELETE 数据，假设 Iceberg Writer 在写入 a1 和 b1 的 INSERT 数据后，就关闭并新开启了一个文件，那么此时写入的记录 c1 和对应的行号会被记录在内存中。此时 Writer 接收到 user\_id=c1 的数据后，便能直接从内存中找到 user\_id=c1 的数据是在 fileA 中的第一行，此时写下一个 Position Delete File；而 user\_id=a1 的 DELETE 数据，由于文件已经关闭，内存中没有记录其信息，所以写下一个 Equality Delete File。

## Sequence Number

引入 DELETE 操作后，如果在读取时进行合并，则涉及到一个问题，如果用户对同一个 equality\_id 的数据进行插入、删除、再插入，那么读取时该如何保证把第一次插入的数据给删掉，读取第二次插入的数据？

这里的处理方式是将 Data File 和 Delete File 放在一起按写入顺序编号，在读取时，DELETE 只对小于当前 Sequence Number 的 Data File 生效。如果遇到相同记录的并发写入的时候怎么办？这里就要利用 Iceberg 自身的事务机制了，Iceberg Writer 在写入前会检查相关 meta 以及 Sequence Number，如果写入后不符合预期则会采取乐观锁的形式进行重试。

## Schema Evolution

Iceberg 的 schema evolution 是其特色之一，支持以下操作：

增加字段

删除字段

重命名字段

修改字段

改变字段顺序

关于 schema 的变更也依赖上面文件结构，由于每次写入时，都会产生 snapshot -> manifest -> data file 的层级，同样，读取时也会从 snapshot 开始读取并路由到对应的底层 data file。所以 Iceberg 只需要每次写入时在 manifest 中记录下 schema 的情况，并在读取时进行对应的转换即可。

# 架构