Flink + Iceberg + 对象存储，构建数据湖方案

原创 孙伟@Dell Flink 中文社区 7月8日

摘要：本文整理自 Dell 科技集团高级软件研发经理孙伟在 4 月 17 日 上海站 Flink Meetup 分享的《Iceberg 和对象存储构建数据湖方案》。内容包括：

1. 数据湖和 Iceberg 简介
2. 未来规划
3. 演示方案
4. 存储优化的一些思考

Tips：点击文末**「阅读原文」**即可查看更多技术干货～

🎉 GitHub 地址 🎉

欢迎大家给 Flink 点赞送 star~

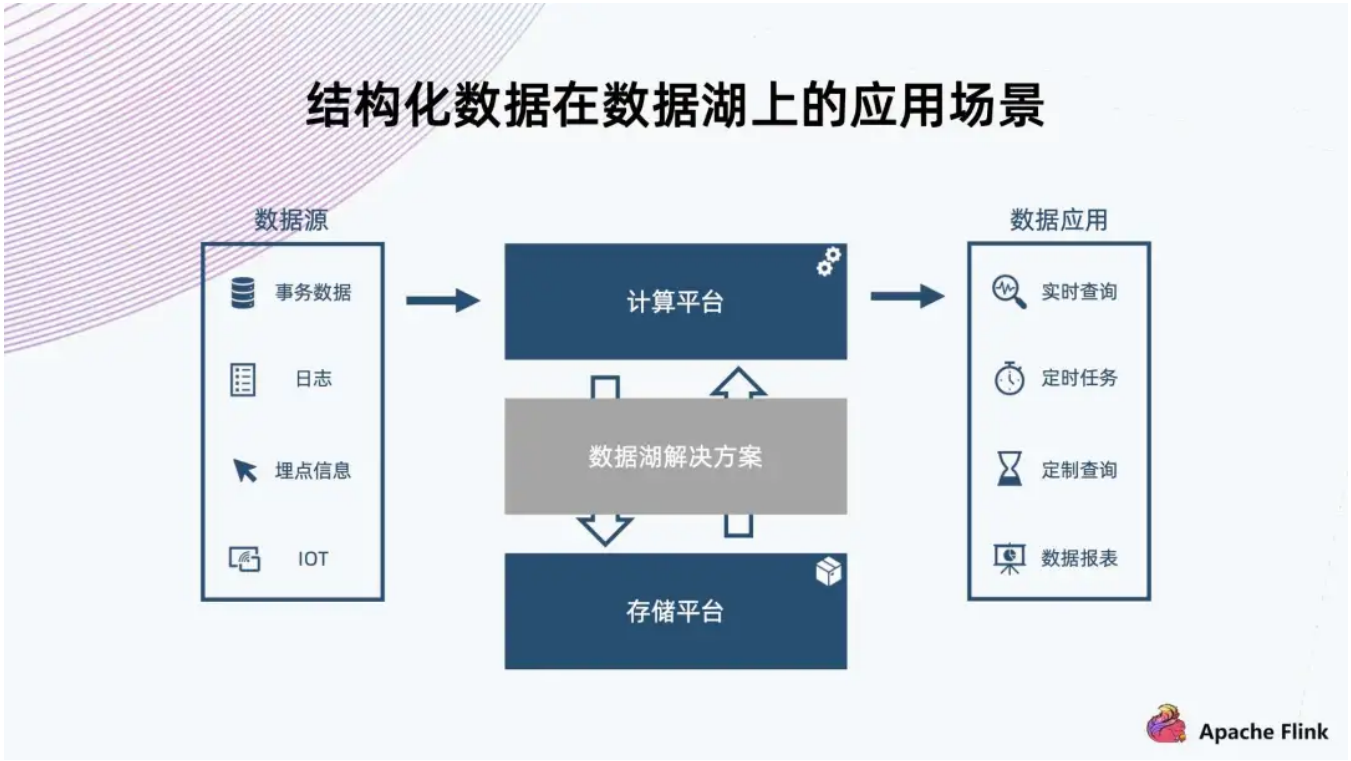
一、数据湖和 Iceberg 简介

1. 数据湖生态

如上图所示，对于一个成熟的数据湖生态而言：

- 首先我们认为它底下应具备海量存储的能力，常见的有对象存储，公有云存储以及 HDFS；
- 在这之上，也需要支持丰富的数据类型，包括非结构化的图像视频，半结构化的 CSV、XML、Log，以及结构化的数据库表；
- 除此之外，需要高效统一的元数据管理，使得计算引擎可以方便地索引到各种类型数据来做分析。
- 最后，我们需要支持丰富的计算引擎，包括 Flink、Spark、Hive、Presto 等，从而方便对接企业中已有的一些应用架构。

2. 结构化数据在数据湖上的应用场景



上图为一个典型的数据湖上的应用场景。

数据源上可能会有各种数据，不同的数据源和不同格式。比如说事物数据，日志，埋点信息，IOT 等。这些数据经过一些流然后进入计算平台，这个时候它需要一个结构化的方案，把数据组织放到一个存储平台上，然后供后端的数据应用进行实时或者定时的查询。

这样的数据库方案它需要具备哪些特征呢?

- 首先，可以看到数据源的类型很多，因此需要支持比较丰富的数据 Schema 的组织；

- 其次，它在注入的过程中要支撑实时的数据查询，所以需要 ACID 的保证，确保不会读到一些还没写完的中间状态的脏数据；

- 最后，例如日志这些有可能临时需要改个格式，或者加一列。类似这种情况，需要避免像传统的数仓一样，可能要把所有的数据重新提出来写一遍，重新注入到存储；而是需要一个轻量级的解决方案来达成需求。

Iceberg 数据库的定位就在于实现这样的功能，于上对接计算平台，于下对接存储平台。

3. 结构化数据在数据湖上的典型解决方案

对于数据结构化组织，典型的解决方式是用数据库传统的组织方式。

如上图所示，上方有命名空间，数据库表的隔离；中间有多个表，可以提供多种数据 Schema 的保存；底下会放数据，表格需要提供 ACID 的特性，也支持局部 Schema 的演进。

4. Iceberg 表数据组织架构

- 快照 Metadata：表格 Schema、Partition、Partition spec、Manifest List 路径、当前快照等。
- Manifest List：Manifest File 路径及其 Partition，数据文件统计信息。
- Manifest File：Data File 路径及其每列数据上下边界。

- Data File：实际表内容数据，以 Parque，ORC，Avro 等格式组织。

接下来具体看一下 Iceberg 是如何将数据组织起来的。如上图所示：

- 可以看到右边从数据文件开始，数据文件存放表内容数据，一般支持 Parquet、ORC、Avro 等格式；
- 往上是 Manifest File，它会记录底下数据文件的路径以及每列数据的上下边界，方便过滤查询文件；
- 再往上是 Manifest List，它来链接底下多个 Manifest File，同时记录 Manifest File 对应的分区范围信息，也是为了方便后续做过滤查询；
  Manifest List 其实已经表示了快照的信息，它包含当下数据库表所有的数据链接，也是 Iceberg 能够支持 ACID 特性的关键保障。
  有了快照，读数据的时候只能读到快照所能引用到的数据，还在写的数据不会被快照引用到，也就不会读到脏数据。多个快照会共享以前的数据文件，通过共享这些 Manifest File 来共享之前的数据。
- 再往上是快照元数据，记录了当前或者历史上表格 Scheme 的变化、分区的配置、所有快照 Manifest File 路径、以及当前快照是哪一个。
  同时，Iceberg 提供命名空间以及表格的抽象，做完整的数据组织管理。

5. Iceberg 写入流程

上方为 Iceberg 数据写入的流程图，这里用计算引擎 Flink 为例。

- 首先，Data Workers 会从元数据上读出数据进行解析，然后把一条记录交给 Iceberg 存储；
- 与常见的数据库一样，Iceberg 也会有预定义的分区，那些记录会写入到各个不同的分区，形成一些新的文件；
- Flink 有个 CheckPoint 机制，文件到达以后，Flink 就会完成这一批文件的写入，然后生成这一批文件的清单，接着交给 Commit Worker；
- Commit Worker 会读出当前快照的信息，然后与这一次生成的文件列表进行合并，生成一个新的 Manifest List 以及后续元数据的表文件的信息，之后进行提交，成功以后就形成一个新的快照。

6. Iceberg 查询流程

上方为 Iceberg 数据查询流程。

- 首先是 Flink Table scan worker 做一个 scan，scan 的时候可以像树一样，从根开始，找到当前的快照或者用户指定的一个历史快照，然后从快照中拿出当前快照的 Manifest List 文件，根据当时保存的一些信息，就可以过滤出满足这次查询条件的 Manifest File；

- 再往下经过 Manifest File 里记录的信息，过滤出底下需要的 Data Files。这个文件拿出来以后，再交给 Recorder reader workers，它从文件中读出满足条件的 Recode，然后返回给上层调用。

这里可以看到一个特点，就是在整个数据的查询过程中没有用到任何 List，这是因为 Iceberg 完整地把它记录好了，整个文件的树形结构不需要 List，都是直接单路径指向的，因此查询性能上没有耗时 List 操作，这点对于对象存储比较友好，因为对象存储在 List 上面是一个比较耗资源的操作。

7. Iceberg Catalog 功能一览

Iceberg 提供 Catalog 用良好的抽象来对接数据存储和元数据管理。任何一个存储，只要实现 Iceberg 的 Catalog 抽象，就有机会跟 Iceberg 对接，用来组织接入上面的数据湖方案。

如上图所示，Catalog 主要提供几方面的抽象。

- 它可以对 Iceberg 定义一系列角色文件；
- 它的 File IO 都是可以定制，包括读写和删除；
- 它的命名空间和表的操作 (也可称为元数据操作)，也可以定制；
- 包括表的读取 / 扫描，表的提交，都可以用 Catalog 来定制。

这样可以提供灵活的操作空间，方便对接各种底下的存储。

二、对象存储支撑 Iceberg 数据湖

1. 当前 Iceberg Catalog 实现

目前社区里面已经有的 Iceberg Catalog 实现可分为两个部分，一是数据 IO 部分，二是元数据管理部分。

如上图所示，其实缺少面向私有对象存储的 Catalog 实现，S3A 理论上可以接对象存储，但它用的是文件系统语义，不是天然的对象存储语义，模拟这些文件操作会有额外的开销，而我们想实现的是把数据和元数据管理全部都交给一个对象存储，而不是分离的设计。

2. 对象存储和 HDFS 的比较

这里存在一个问题，在有 HDFS 的情况下，为什么还要用对象存储?
如下所示，我们从各个角度将对象存储和 HDFS 进行对比。

总结下来，我们认为：

- 对象存储在集群扩展性，小文件友好，多站点部署和低存储开销上更加有优势；
- HDFS 的好处就是提供追加上传和原子性 rename，这两个优势正是 Iceberg 需要的。

下面对两个存储各自的优势进行简单阐述。

■ 1）比较之：集群扩展性

- HDFS 架构是用单个 Name Node 保存所有元数据，这就决定了它单节点的能力有限，所以在元数据方面没有横向扩展能力。
- 对象存储一般采用哈希方式，把元数据分隔成各个块，把这个块交给不同 Node 上面的服务来进行管理，天然地它元数据的上限会更高，甚至在极端情况下可以进行 rehash，把这个块切得更细，交给更多的 Node 来管理元数据，达到扩展能力。

■ 2）比较之：小文件友好

如今在大数据应用中，小文件越来越常见，并逐渐成为一个痛点。

- HDFS 基于架构的限制，小文件存储受限于 Name Node 内存等资源，虽然 HDFS 提供了 Archive 的方法来合并小文件，减少对 Name Node 的压力，但这需要额外增加复杂度，不是原生的。
  同样，小文件的 TPS 也是受限于 Name Node 的处理能力，因为它只有单个 Name Node。对象存储的元数据是分布式存储和管理，流量可以很好地分布到各个 Node 上，这样单节点就可以存储海量的小文件。

- 目前，很多对象存储提供多介质，分层加速，可以提升小文件的性能。

■ 3）比较之：多站点部署

- 对象存储支持多站点部署
  - 全局命名空间
  - 支持丰富的规则配置

- 对象存储的多站点部署能力适用于两地三中心多活的架构，而 HDFS 没有原生的多站点部署能力。虽然目前看到一些商业版本给 HDFS 增加了多站点负责数据的能力，但由于它的两个系统可能是独立的，因此并不能支撑真正的全局命名空间下多活的能力。

■ 4）比较之：低存储开销

- 对于存储系统来说，为了适应随机的硬件故障，它一般会有副本机制来保护数据。
    - 常见的如三副本，把数据存三份，然后分开保存到三个 Node 上面，存储开销是三倍，但是它可以同时容忍两个副本遇到故障，保证数据不会丢失。
    - 另一种是 Erasure Coding，通常称为 EC。以 10+2 举例，它把数据切成 10 个数据块，然后用算法算出两个代码块，一共 12 个块。接着分布到四个节点上，存储开销是 1.2 倍。它同样可以容忍同时出现两个块故障，这种情况可以用剩余的 10 个块算出所有的数据，这样减少存储开销，同时达到故障容忍程度。
- HDFS 默认使用三副本机制，新的 HDFS 版本上已经支持 EC 的能力。经过研究，它是基于文件做 EC，所以它对小文件有天然的劣势。因为如果小文件的大小小于分块要求的大小时，它的开销就会比原定的开销更大，因为两个代码块这边是不能省的。在极端情况下，如果它的大小等同于单个代码块的大小，它就已经等同于三副本了。

同时，HDFS 一旦 EC，就不能再支持 append、hflush、hsync 等操作，这会极大地影响 EC 能够使用的场景。对象存储原生支持 EC，对于小文件的话，它内部会把小文件合并成一个大的块来做 EC，这样确保数据开销方面始终是恒定的，基于预先配置的策略。

3. 对象存储的挑战：数据的追加上传

在 S3 协议中，对象在上传时需要提供大小。

以 S3 标准为例，对象存储跟 Iceberg 对接时，S3 标准对象存储不支持数据追加上传的接口，协议要求上传文件时提供文件大小。所以在这种情况下，对于这种流式的 File IO 传入，其实不太友好。

■ 1）解决方案一：S3 Catalog 数据追加上传 - 小文件缓存本地/内存

对于一些小文件，流式传入的时候就写入到本地缓存 / 内存，等它完全写完后，再把它上传到对象存储里。

■ 2）解决方法二：S3 Catalog 数据追加上传 - MPU 分段上传大文件

对于大文件，会用到 S3 标准定义的 MPU 分段上传。

它一般分为几个步骤：

- 第一步先创建初始化的 MPU，拿到一个 Upload ID，然后给每一个分段赋予一个 Upload ID 以及一个编号，这些分块就可以并行上传；
- 在上传完成以后，还需要一步 Complete 操作，这样相当于通知系统，它会把基于同一个 Upload ID 以及所有的编号，从小到大排起来，组成一个大文件；
- 把机制运用到数据追加上传场景，常规实现就是写入一个文件，把文件缓存到本地，当达到分块要求大小时，就可以把它进行初始化 MPU，把它的一个分块开始上传。后面每一个分块也是一样的操作，直到最后一个分块上传完，最后再调用一个完成操作来完成上传。

MPU 有优点也有缺点：

- 缺点是 MPU 的分片数量有上限，S3 标准里可能只有 1 万个分片。想支持大文件的话，这个分块就不能太小，所以对于小于分块的文件，依然是要利用前面一种方法进行缓存上传；
- MPU 的优点在于并行上传的能力。假设做一个异步的上传，文件在缓存达到以后，不用等上一个分块上传成功，就可以继续缓存下一个，之后开始上传。当前面注入的速度足够快时，后端的异步提交就变成了并行操作。利用这个机制，它可以提供比单条流上传速度更快的上传能力。

4. 对象存储的挑战：原子提交

下一个问题是对象存储的原子提交问题。

前面提到在数据注入的过程中，最后的提交其实分为几步，是一个线性事务。首先它要读到当前的快照版本，然后把这一次的文件清单合并，接着提交自己新的版本。这个操作类似于我们编程里常见的 “i=i+1”，它不是一个原子操作，对象存储的标准里也没有提供这个能力。

上图是并发提交元信息的场景。

- 这里 Commit Worker 1 拿到了 v006 版本，然后合并自己的文件，提交 v007 成功。
- 此时还有另一个 Commit Worker 2，它也拿到了 v006，然后合并出来，且也要提供 v007。此时我们需要一个机制告诉它 v007 已经冲突，不能上传，然后让它自己去 Retry。Retry 以后取出新的 v007 合并，然后提交给 v008。

这是一个典型的冲突场景，这里需要一套机制，因为如果它不能检测到自己是一个冲突的情况的话，再提交 v007 会把上面 v007 覆盖，会导致上一次提交的所有数据都丢失。

如上图所示，我们可以使用一个分布式锁的机制来解决上述问题。

- 首先，Commit Worker 1 拿到 v006，然后合并文件，在提交之前先要获取这一把锁，拿到锁以后判断当前快照版本。如果是 v006，则 v007 能提交成功，提交成功以后再解锁。
- 同样，Commit Worker 2 拿到 v006 合并以后，它一开始拿不到锁，要等 Commit Worker 1 释放掉这个锁以后才能拿到。等拿到锁再去检查的时候，会发现当前版本已经是 v007，与自己的 v007 有冲突，因此这个操作一定会失败，然后它就会进行 Retry。

这是通过锁来解决并发提交的问题。

5. Dell EMC ECS 的数据追加上传

基于 S3 标准的对象存储和 Iceberg 问题的解决方案存在一些问题，例如性能损失，或者需要额外部署锁服务等。

Dell EMC ECS 也是个对象存储，基于这个问题有不一样的解答，它基于 S3 的标准协议有一些扩展，可以支持数据的追加上传。

它的追加上传与 MPU 不同的地方在于，它没有分块大小的限制。分块可以设置得比较小一点，上传后内部就会串联起来，依然是一个有效的文件。

追加上传和 MPU 这两者可以在一定程度上适应不同的场景。

MPU 有加速上传能力，追加上传在速度在不是很快的情况下，性能也是足够用，而且它没有 MPU 的初始化和合并的操作，所以两者在性能上能够适应不同场景进行使用。

6. Dell EMC ECS 在并发提交下的解决方案

ECS 对象存储还提供了一个 If-Match 的语义，在微软的云存储以及谷歌的云存储上都有这样一个接口能力。

- If-Match 就是说在 Commit Worker 1 提交拿到 v006 的时候，同时拿到了文件的 eTag。提交的时候会带上 eTag，系统需要判断要覆盖文件的 eTag 跟当前这个文件真实 eTag 是否相同，如果相同就允许这次覆盖操作，那么 v007 就能提交成功；

- 另一种情况，是 Commit Worker 2 也拿到了 v006 的 eTag，然后上传的时候发现拿到 eTag 跟当前系统里文件不同，则会返回失败，然后触发 Retry。

这个实现是和锁机制一样的效果，不需要外部再重新部署锁服务来保证原子提交的问题。

7. S3 Catalog - 统一存储的数据

回顾一下，上方我们解决了文件 IO 中上传数据 IO 的问题，和解决了元数据表格的原子提交问题。

解决这些问题以后，就可以把数据以及元数据的管理全部都交到对象存储，不再需要额外部署元数据服务，做到真正统一数据存储的概念。

三、演示方案

如上所示，演示方案用到了 Pravega，可以简单理解为 Kafka 的一个替代，但是对它进行了性能优化。

在这个例子中，我们会把数据注入 Pravega 的流里，然后 Flink 会从 Pravega 中读出数据进行解析，然后存入 Iceberg 组织。Iceberg 利用 ECS Catalog，直接对接对象存储，这里面没有任何其他部署，最后用 Flink 读出这个数据。

四、存储优化的一些思考

上图为当前 Iceberg 支持的数据组织结构，可以看到它直接 Parquet 文件存在存储里面。

我们的想法是如果这个湖跟元数据的湖其实是一个湖，有没有可能生成的 Parquet 文件跟源文件存在很大的数据冗余度，是否可以减少冗余信息的存储。

比如最极端的情况，源文件的一个信息记录在 Iceberg 中，就不存这个 Parquet 数据文件。当要查询的时候，通过定制 File IO，让它根据原文件在内存中实时生成一个类似于 Parquet 的格式，提交给上层应用查询，就可以达到一样的效果。

但是这种方式，局限于对存储的成本有很高的要求，但是对查询的性能要求却不高的情况。能够实现这个也要基于 Iceberg 好的抽象，因为它的文件元数据和 File IO 都是抽象出来的，可以把源文件拆进去，让它以为这是一个 Parquet 文件。

进一步思考，能否优化查询性能，同时节省存储空间?

比如预计算一下，把源文件某些常用的列拿出来，然后统计信息到 Iceberg 中，在读的时候利用源文件和云计算的文件，可以很快查询到信息，同时又节省了不常用的数据列存储空间。

这是比较初步的想法，如果能够实现，则用 Iceberg 不仅可以索引结构化的 Parquet 文件格式，甚至可以索引一些半结构化、结构化的数据，通过临时的计算来解决上层的查询任务，变成一个更完整的 Data Catalog。

另外～《Apache Flink-实时计算正当时》电子书重磅发布，本书将助您轻松 Get Apache Flink 1.13 版本最新特征，同时还包含知名厂商多场景 Flink 实战经验，学用一体，干货多多！快扫描下方二维码获取吧～

（本次为抢鲜版，正式版将于 7 月初上线）

更多 Flink 相关技术交流，可扫码加入社区钉钉大群～

▼ 关注「**Flink 中文社区**」，获取更多技术干货 ▼

喜欢此内容的人还喜欢

OLAP数仓入门：基础篇

浪尖聊大数据

是的，我们不用 Kubernetes

InfoQ

Apache APISIX 在移动云对象存储 EOS 的应用与实践

InfoQ