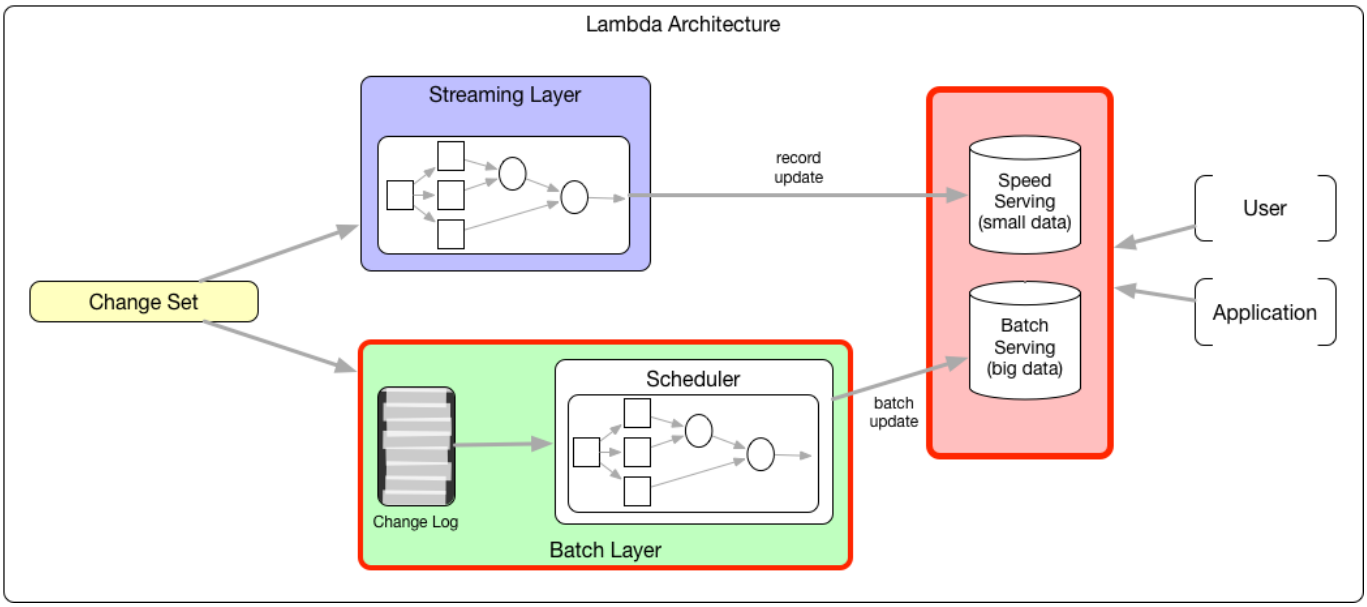
Apache Hudi: Uber 开源的大数据增量处理框架

随着 Apache Parquet 和 Apache ORC 等存储格式以及 Presto 和 Apache Impala 等查询引擎的发展，Hadoop 生态系统有可能成为一个面向几分钟延迟工作负载的通用统一服务层。但是，为了实现这一点，需要在 Hadoop 分布式文件系统(HDFS)中实现高效、低延迟的数据摄取和数据准备。

为了解决这个问题，Uber 构建了Hudi（被称为“hoodie”），这是一个增量处理框架（incremental processing），可以以低延迟和高效率的方式支持所有业务关键数据管道。事实上，Uber 在2017年开放了源代码（https://github.com/uber/hudi），供其他人使用和构建，并且在2019年4月 Uber 向 Apache 软件基金会提交开源大数据存储库 Hudi。但是在深入讨论 Hudi 之前，我们首先讨论一下为什么将 Hadoop 作为统一的服务层是一个不错的想法。

动机

Lambda 架构是一种常见的数据处理架构，它提出了具有流式和批处理层的双重计算。每隔几个小时，就会启动一个批处理过程来计算准确的业务状态，并将批量更新加载到服务层。同时，流处理层计算并提供相同的状态，以避免上述的多小时延迟。但是，在被更精确的批处理计算状态覆盖之前，此状态只是一个近似状态。因为流处理和批处理的状态略有不同，所以需要批处理和流处理提供不同的服务层，并在这个上面再做合并抽象，或者使用一个相当复杂的服务系统(比如Druid)，这个系统在记录级更新和批量加载方面表现得相当好。

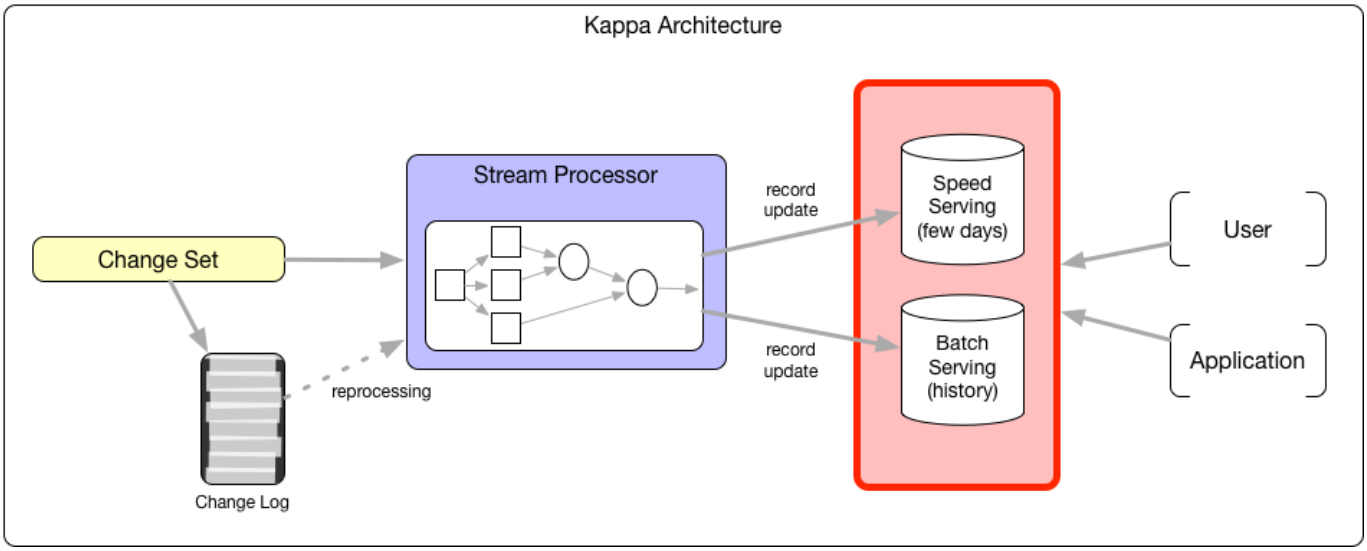


如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

当质疑是否需要单独的批处理层时，Kappa

体系结构认为流处理引擎可以作为计算的通用解决方案。 从广义上讲，所有数据计算都可以描述为生产者生产一个数据流，而消费者不断的逐条迭代消费这个流中的记录（比如 Volcano Iterator model）。这个特点使得我们可以在流式处理层通过增加并行性和资源来对有状态的业务数据进行重新处理。这类系统可以依靠有效的检查点（checkpoint) 和大量的状态管理来让流式处理的结果不再只是一个近似值。这个模型被应用于很多的数据摄取任务（ingest pipelines）。尽管如此，虽然批处理层在这个模型中被去掉了，但是在服务层仍然存在两个问题。

如今很多流式处理引擎都支持行级的数据处理，这就要求我们的服务层也需要能够支持行级更新的能力。通常，这类系统并不能对分析类的查询扫描优化到这个地步，除非我们在内存中缓存大量记录（如Memsql）或者有强大的索引支持（如ElasticSearch）。这些系统为了获得数据摄取和扫描的性能往往需要增加成本和牺牲服务的可扩展性。出于这个原因，这类服务系统的数据驻留的能力往往是有限的，从时间上可能30~90天，从总量上来说几个TB的数据就是他们的极限了。对于历史数据的分析又会被重新定向到时延要求不那么高的HDFS上。



数据摄取延迟、扫描性能、计算资源和操作复杂性之间的基本权衡是不可避免的。但是对于能够容忍大约10分钟延迟的工作负载，如果有一种更快速的方式在 HDFS
中摄取和准备数据，则不需要单独的 Speed serving
层。这统一了服务层，并显著降低了系统总体的复杂性和资源使用。

然而，要使HDFS成为统一的服务层，它不仅需要存储变更集的日志（一个日志记录系统），而且还需要支持由有意义的业务度量划分的压缩的、去重复的业务状态。这种类型的统一服务层需要具备以下特点：

- 能够在 HDFS 大型数据集上应用更新（apply mutations）
- 数据存储层需要为分析扫描进行优化（列式文件格式）
- 能够有效地链接和传播更新到建模数据集

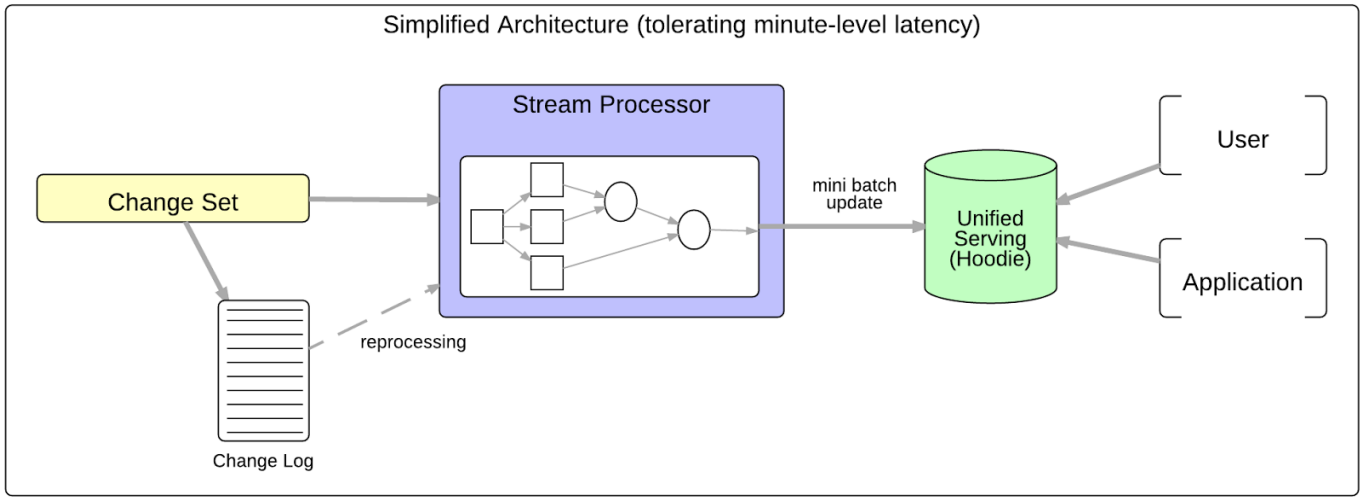
被压缩的业务状态变更是无法避免的，即使我们以事件时间（Event time）作为业务分区字段。

由于延迟到达的数据以及事件时间和处理时间之间的差异，仍然会导致对许多旧分区进行更新。即使分区字段是处理时间，也可能存在更新的情况，比如出于审计、安全方面的考虑，需要清除数据。

Hudi 介绍: Hi, Hudi!

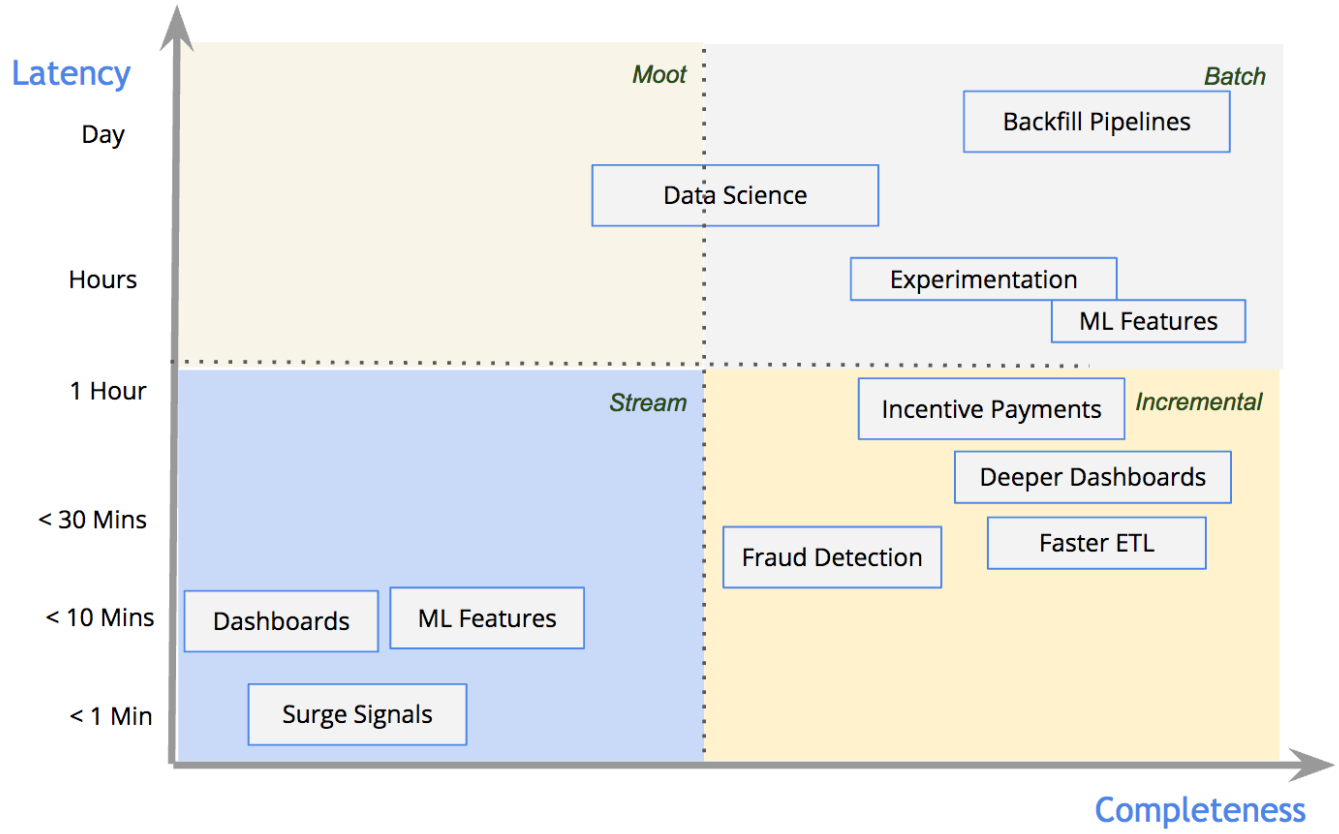
下面让我们来看看
Hudi，这是一个增量数据处理的框架，它解决了我们在前面说的各种问题。简而言之，Hudi (Hadoop Upsert Delete and I
ncremental)是一种分析和扫描优化的数据存储抽象，可在几分钟之内将变更应用于 HDFS 中的数据集中，并支持多个增量处理系统处理数据。

Hudi 数据集通过自定义的 InputFormat 与当前 Hadoop 生态系统(包括 Apache Hive、Apache Parquet、Presto 和 Apache Spark)集成，使得该框架对最终用户来说是无缝的。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

Google 的 DataFlow 模型基于数据管道的延迟和完整性保证来对数据管道进行分类。下图展示了 Uber 的各种用户场景按照这个逻辑进行的划分，以及每种流水通常采用的处理方式：



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

对于少数真正需要1分钟延迟和具有简单业务度量的展示用例，我们依赖于记录级流处理。对于传统的批处理用例，如机器学习，我们依赖于批处理。对于包含复杂 join
或者重要数据处理的近实时场景，我们基于 Hudi
以及它的增量处理原语来获得两全其美的结果。想要了解更多关于 Hudi 支持的用例，您可以在 Github 上查看相关文档。

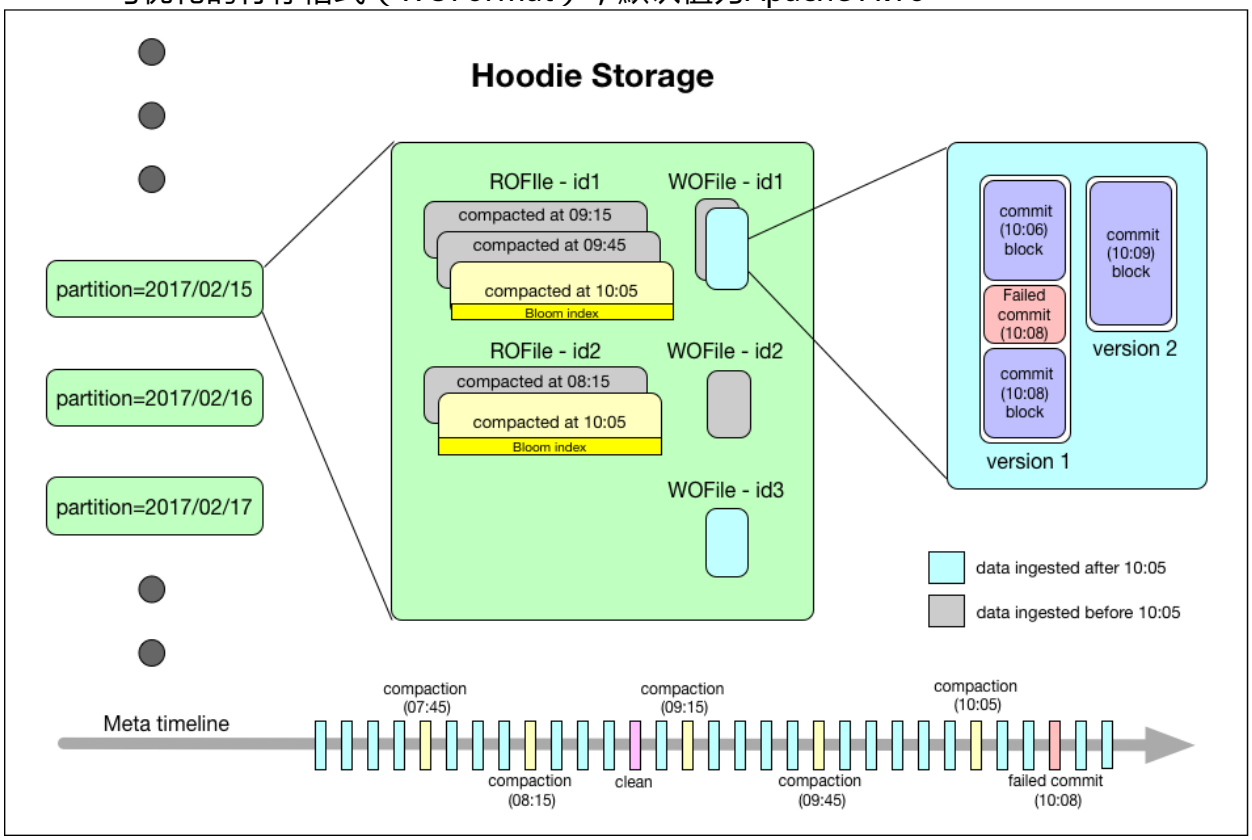
Hudi 存储引擎

Hudi 将数据集组织到 basepath 下的分区目录结构中，类似于传统的 Hive
表。数据集被分成分区，分区是包含该分区的数据文件的目录。每个分区由其相对于 basepath 的 partitionpath
唯一标识。在每个分区中，记录被分布到多个数据文件中。每个数据文件都由唯一的 fileId
和生成该文件的 commit 标识。对于更新，多个数据文件可以共享同一个
fileId，但对应于不同的 commit。

每条记录都由一个记录键（record key）惟一标识，并映射到一个
fileId。一旦记录的第一个版本被写入文件，记录键和 fileId
之间的映射就永久不变。简而言之，fileId
标识一组文件，而这些文件包含所有记录的所有版本数据。

Hudi 的存储引擎由三个不同的部分组成：

- Metadata：Hudi 以时间轴（timeline）的形式将数据集上的各项操作对应的元数据维护起来，从而支持数据集的即时视图，这部分元数据存储于根目录（basepath）下的元数据目录中。下面我们简单介绍一下时间轴中对应的三种操作：
  - 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。Commits 由单调递增的时间戳标识，表示写操作的开始；
  - 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：协调 Hudi
    中不同数据结构的后台活动，比如将基于行更新的文件转换成列式存储格式。
- Index：Hudi
  维护了一个索引，以便在记录键已经存在的情况下，快速地将传入的记录键映射到 fileId。索引实现是可插拔的，以下是目前可用的选项：
  - Bloom filter
    ：存储在每个数据文件页脚中，默认就是用这个，因为不依赖任何外部系统。数据和索引始终保持一致。
  - Apache HBase ：可高效地查找一小批 key。在索引标记期间（index tagging），这个索引实现可能会快几秒钟。
- Data：Hudi 以两种不同的存储格式存储所有摄入的数据。但实际使用的存储格式是可插拔的，但所选的存储格式需要以下特征：
  - 扫描优化的列存格式（ROFormat），默认值为 Apache Parquet
  - 写优化的行存格式（WOFormat），默认值为Apache Avro



如果想

及时了解Spar
k、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

Optimization

Hudi 对 HDFS 的使用模式进行了优化。Compaction
是将数据从写优化格式转换为读优化格式的关键操作。由于 Compaction
操作的基本并行单位是重写单个 fileId，所以 Hudi 确保了所有的数据文件大小和 HDFS
块大小对齐，以平衡压缩并行性、查询扫描并行性和HDFS中的文件总数，Compaction
操作也是可插拔的，可以扩展为合并不频繁更新的老的数据文件已进一步减少文件总数。

数据写路径（Ingest Path）

Hudi 是一个 Spark 库，以流摄取作业的形式运行，并以小批量的方式摄取数据(通常是一到两分钟)。但是，根据延迟需求和资源协商时间，也可以使用 Apache Oozie 或 Apache Airflow
将摄取的作业作为离线任务运行。

以下是默认配置下 Hudi 的写入路径：

- Hudi 从相关分区中的所有 parquet 文件加载 Bloom filter 索引，并通过传入的 key 值映射到对应的文件来标记是更新还是插入。这里的连接操作可能由于输入数据批次大小，分区的分布或者单个分区下的文件数问题导致数据倾斜。通过对连接字段进行范围分区以及新建子分区的方式处理，以避免 Spark 某些低版本中处理 Shuffle 文件时的 2GB
限制的问题。
- Hudi 对每个分区的插入进行分组，分配一个新的
fileId，并将其追加到相应的日志文件中，直到日志文件达到 HDFS
块大小。一旦达到块大小，Hudi 就创建另一个
fileId，并对该分区中的所有插入重复这个过程，直到所有的数据都被插入。
    - 调度程序每隔几分钟就会启动一个有时间限制的压缩任务，它会生成一个按优先级排序的压缩列表，并将 fileId 的所有 avro 文件压缩到当前 parquet
文件中，并创建该 Parquet 文件的下一个版本。
    - Compaction
操作是异步进行的，锁定要压缩的特定日志版本，并以新的日志记录更新到对应 fileId 中，锁维护在Zookeeper中。
    - Compaction 操作的优先级顺序由被压缩的日志数据大小决定。在每次压缩迭代过程中，首先压缩日志量最大的文件，最后压缩较小的日志文件，因为重新编写 parquet 文件的成本不会根据文件的更新次数进行分摊。
- 在操作更新记录的时候，如果存在一个 fileId, Hudi
会将其追加到相应的日志文件中；如果不存在，则创建一个日志文件。
- 如果摄取作业成功，Hudi 会在时间轴中记录提交，该时间轴自动将一个 inflight
文件重命名为 commit 文件，并记录有关分区和创建的 fileId 版本的详细信息。

Optimization

如前所述，Hudi 会尽可能使文件大小与底层块大小保持一致。根据列式压缩的效率和要压缩的分区中的数据量，压缩仍然会创建小型 parquet 文件。因为对分区的插入操作会是以对现有小文件的更新来进行的，所有这些小文件的问题最终会被一次次的迭代不断修正。最终，文件大小会不断增长直到与HDFS块大小一致。

故障恢复

如果一个摄取作业由于间歇性错误而失败，Spark 将重试计算 RDD
并自动解决这个问题。如果失败的次数超过了 Spark 的最大尝试次数，则摄取作业失败，下一次迭代将再次尝试摄取相同的批处理。以下指出两个重要区别:

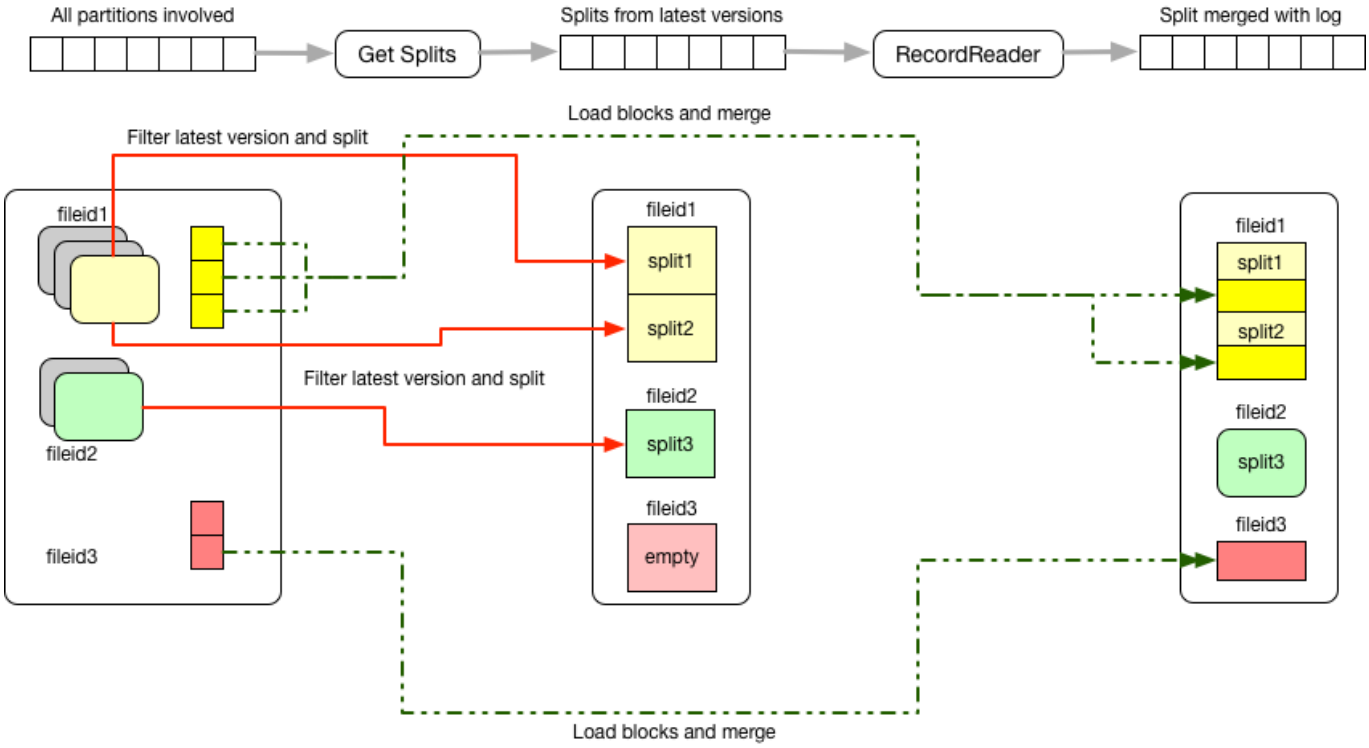
- 摄取失败可能在日志文件中生成包含部分数据的 avro 块：这个问题通过在 commit 元数据中存储对应数据块的起始偏移量和日志文件版本来解决。当读取日志文件时，偶尔发生的部分写入的数据块会被跳过，且会从正确的位置开始读取avro文件。
- Compaction 过程失败会生产包含部分数据的 parquet
  文件：这个问题在查询阶段被解决，通过 commit
  元数据进行文件版本的过滤。查询阶段只会读取最新的完成的 compaction
  后的文件。这些失败的 compaction 文件会在下一个 compaction 周期被回滚。

查询路径

提交的时间轴支持在 HDFS 中对相同数据进行读取优化的视图和实时视图；这些视图允许客户在数据延迟和查询执行时间之间进行选择。Hudi 为这些视图提供了自定义的
InputFormat，并包含了一个 Hive 注册模块，该模块将这两个视图注册为Hive metastore
表。这两种输入格式都能够识别 fileId 和提交时间，并可以筛选出最新提交的文件。然后，Hudi 会基于这些数据文件生成输入分片供查询使用。InputFormat 详情如下:

- HoodieReadOptimizedInputFormat:
  提供扫描优化的视图，筛选所有的日志文件并获取最新版本的 parquet 压缩文件
- HoodieRealtimeInputFormat: 提供一个实时的视图，除了会获取最新的 parquet
  压缩文件之外，还提供一个 RecordReader 以合并与 parquet 文件相关的日志文件。

这两种 InputFormats 都扩展了MapredParquetInputFormat 和
VectorizedParquetRecordReader，因此所有针对 parquet 文件的优化依然被保留。通过依赖 hoodie-hadoop-mr 类库，Presto 和 Spark SQL可以对 Hudi 格式 的Hive Metastore
表做到开箱即用。



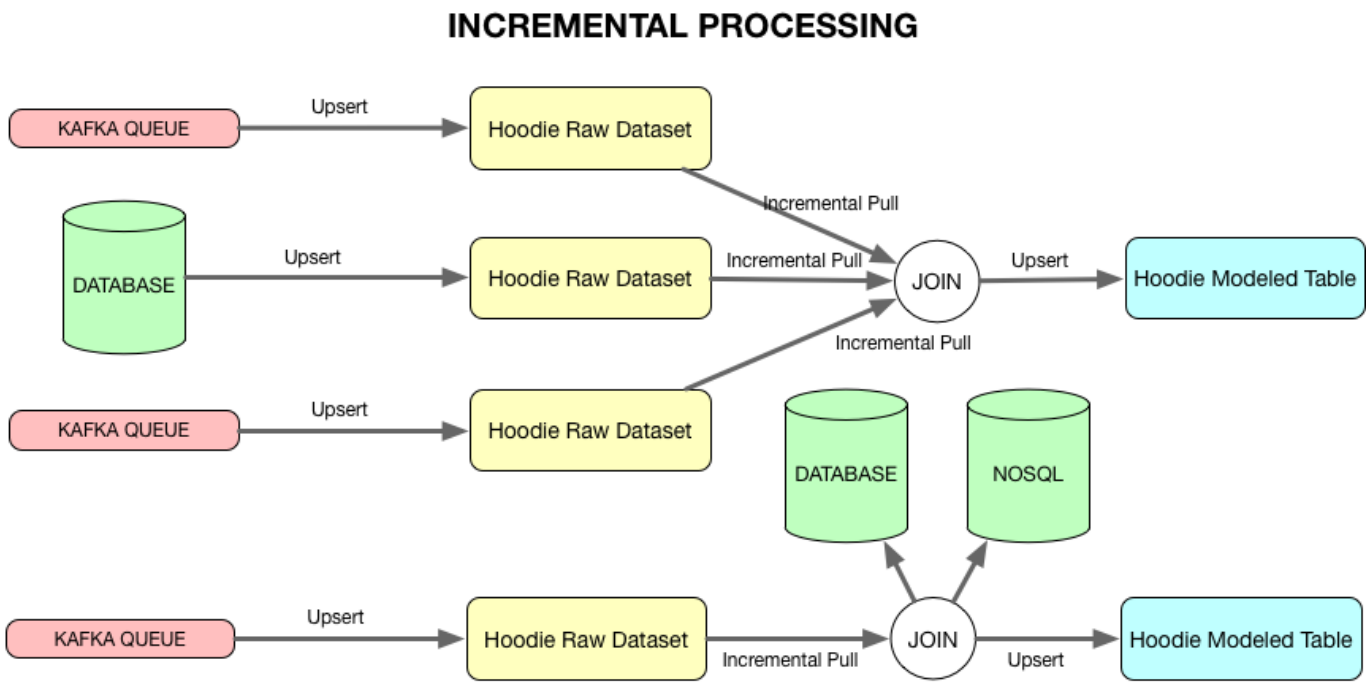
如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

增量处理（Incremental Processing）

如前所述，需要在 HDFS 中处理和提供已建模的表，以便 HDFS
成为统一的服务层。构建低延时的数据模型表需要能够链接 HDFS 数据集的增量处理。由于 Hudi 在元数据中维护了每次提交的提交时间以及对应的文件版本，使得我们可以基于起始时间戳和结束时间戳从特定的 Hudi 数据集中提取增量的变更数据集。

这个过程基本上与普通的查询大致相同，只是选取特定时间范围内的文件版本进行读取而不是选最新的，提交时间会最为过滤条件被谓词下推到文件扫描阶段。这个增量结果集也受到文件自动清理的影响，如果某些时间范围内的文件被自动清理掉了，那自然也是不能被访问到了。

这样使得我们可以基于 watermark 做 stream-to-stream joins 和 stream-to-dataset joins
并对存储在 HDFS 中的建模表进行计算和 upsert 操作。



如果想及时了
解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

本文翻译自：[Hudi: Uber Engineering's Incremental Processing Framework on Apache Hadoop]