60TB 数据量的作业从 Hive 迁移到 Spark 在 Facebook 的实践

Facebook 经常使用分析来进行数据驱动的决策。在过去的几年里，用户和产品都得到了增长，使得我们分析引擎中单个查询的数据量达到了数十TB。我们的一些批处理分析都是基于 Hive 平台（Apache Hive 是 Facebook 在2009年贡献给社区的）和 Corona（ Facebook 内部的 MapReduce 实现）进行的。Facebook 还针对包括 Hive 在内的多个内部数据存储，继续增加了其 Presto 的 ANSI-SQL 查询的覆盖范围。Facebook 内部还支持其他类型的分析，如图计算、机器学习（Apache Giraph）和流处理（如 Puma、Swift 和 Stylus)。

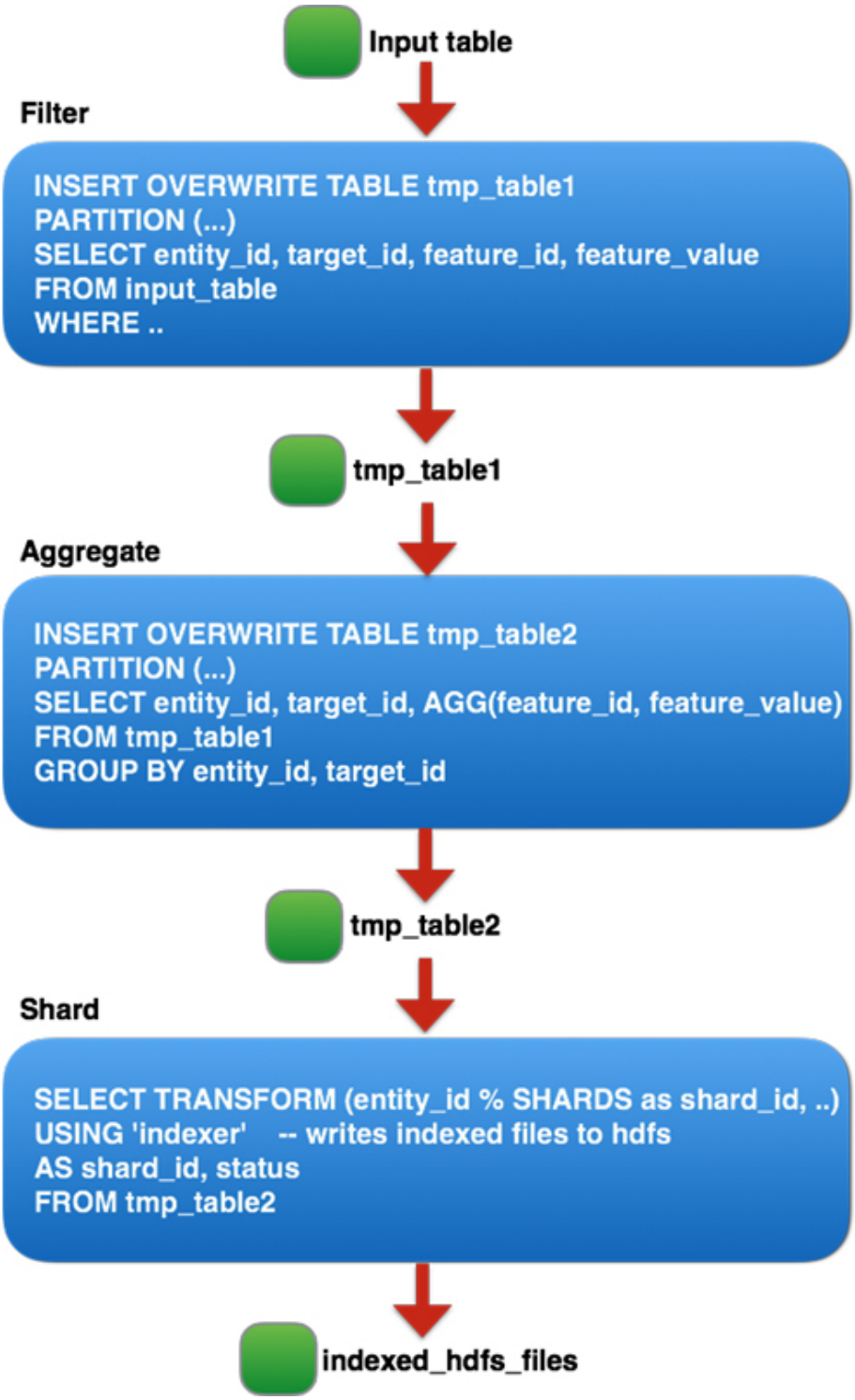
尽管 Facebook 提供的服务涵盖了分析领域的广泛领域，但我们仍在不断地与开源社区互动，以分享我们的经验，并向他人学习。Apache Spark 于2009年由加州大学伯克利分校（UC-Berkeley）的 Matei Zaharia 创办，并于2013年贡献给 Apache。它是目前增长最快的数据处理平台之一，因为它能够支持流处理、批处理、命令式(RDD)、声明式(SQL)、图计算和机器学习用例，所有这些都在相同的 API 和底层计算引擎中。Spark 可以有效地利用大量内存，跨整个管道（pipelines）优化代码，并跨任务（tasks）重用 jvm 以获得更好的性能。Facebook 认为 Spark 已经成熟到可以在许多批处理用例中与 Hive 进行比较的地步。在本文的后面部分，将介绍 Facebook 使用 Spark 替代 Hive 的经验和教训。

用例：为实体排序（entity ranking）做特性准备

实时实体排名在 Facebook 有着多种使用场景。对于一些在线服务平台，原始的特性值是使用 Hive 离线生成的，并将生成的数据加载到这些实时关联查询系统中。这些 Hive 作业是数年前开发的，占用了大量的计算资源，并且难以维护，因为这些作业被拆分成数百个 Hive 小作业。为了使得业务能够使用到新的特征数据，并且让系统变得可维护，我们开始着手将这些作业迁移到 Spark 中。

以前的 Hive 作业实现

基于 Hive 的作业由三个逻辑阶段组成，每个阶段对应数百个由 entity_id 分割的较小 Hive 作业，因为为每个阶段运行较大的 Hive 作业不太可靠，并且受到每个作业的最大任务数限制。具体如下：



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

以上三个逻辑阶段可以概括如下：

- 过滤掉非生产需要的特性和噪音；
- 对每个(entity_id、target_id)对进行聚合；
- 将表分为 N 个分片，并对每个切分通过自定义 UDF

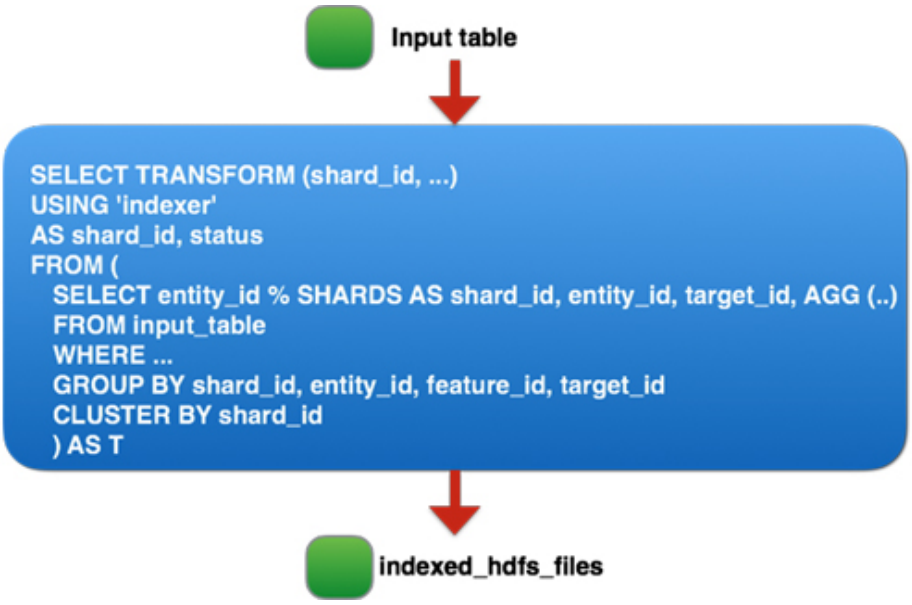
生成一个用于在线查询的自定义索引文件。

基于 Hive 构建索引的作业大约需要运行三天。管理起来也很有挑战性，因为这条管道包含数百个分片作业，因此很难进行监控。没有简单的方法来衡量作业的整体进度或计算 ETA。考虑到现有 Hive 作业的上述局限性，我们决定尝试使用 Spark 来构建一个更快、更易于管理的作业。

Spark 实现

如果使用 Spark 全部替换上面的作业可能会很慢，并且很有挑战性，需要大量的资源。所以我们首先将焦点投入在 Hive
作业中资源最密集的部分：第二阶段。我们从50GB的压缩输入样本开始，然后逐步扩展到 300 GB、1 TB 和20 TB。在每次增加大小时，我们都解决了性能和稳定性问题，但是尝试 20 TB 时我们发现了最大改进的地方。

在运行 20 TB
的输入时，我们发现由于任务太多，生成了太多的输出文件（每个文件的大小大约为100 MB）。在作业运行的10个小时中，有3个小时用于将文件从 staging 目录移动到 HDFS
中的最终目录。最初，我们考虑了两个方案：要么改进 HDFS
中的批量重命名以支持我们的用例；要么配置 Spark 以生成更少的输出文件（这一阶段有大量的任务——70,000个）。经过认真思考，我们得到了第三种方案。由于我们在作业的第二步中生成的 tmp_table2 表是临时的，并且只用于存储作业的中间输出。最后，我们把上面 Hive
实现的三个阶段的作业用一个 Spark 作业表示，该作业读取 60 TB 的压缩数据并执行 90 TB的 shuffle 和排序，最后的 Spark job 如下:



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

我们如何扩展 Spark 来完成这项工作？

当然，在如此大的数据量上运行单个 Spark 作业在第一次尝试甚至第十次尝试时都不会起作用。据我们所知，这是生产环境中 shuffle 数据量最大的 Spark 作业（Databricks 的 PB 级排序是在合成数据上进行的）。我们对 Spark 内核和应用程序进行了大量的改进和优化，才使这项工作得以运行。这项工作的好处在于，其中许多改进都适用于 Spark 的其他大型工作负载，并且我们能够将所有工作重新贡献给开源 Apache Spark 项目 - 有关更多详细信息，请参见下面相关的 JIRA。下面我们将重点介绍将一个实体排名作业部署到生产环境的主要改进。

可靠性修复（Reliability fixes）

处理节点频繁重启

为了可靠地执行长时间运行的作业，我们希望系统能够容错并从故障中恢复（主要是由于正常维护或软件错误导致的机器重新启动）。虽然 Spark 最初的设计可以容忍机器重动，但我们还是发现了各种各样的 bug/问题，我们需要在系统正式投入生产之前解决这些问题。

- 使得 PipedRDD 容忍节点重启（SPARK-13793）：PipedRDD 之前在处理节点重启设计不够健壮，当它获取数据失败时，这个作业就会失败。我们重新设计了 PipedRDD，使得它能够友好的处理这种异常，并且从这种类型的异常中恢复。
- 最大的获取失败次数可配置（SPARK-13369）：对于长期运行的作业而言，由于计算机重动而导致获取失败的可能性大大增加。 在 Spark 中每个阶段允许的最大获取失败次数是写死的，因此，当达到最大失败次数时，作业通常会失败。我们做了一个更改，使其变得可配置，并将这个参数的值从 4 增加到 20，使得作业对于 fetch 失败更加健壮。
- Less disruptive cluster restart：长时间运行的作业应该能够在集群重启后继续运行，这样我们就不会浪费到目前为止完成的所有处理。Spark 的可重启 shuffle service 让我们在节点重启后保留 shuffle 文件。最重要的是，我们在 Spark driver 中实现了能够暂停任务调度的功能，这样作业就不会因为集群重启而导致任务失败。

其他可靠性修复

- Unresponsive driver（SPARK-13279）：Spark driver 添加任务会进行一项时间复杂度为 $O(N^2)$ 的操作，这可能会导致其被卡住，最终导致作业被 killed。我们删除这个不必要的 $O(N^2)$ 操作来解决这个问题。
- Excessive driver speculation：我们发现，Spark driver 在管理大量任务时，会花费了大量时间进行推测（speculation）。在短期内，在运行这个作业时我们禁止了 speculation。我们目前正在对 Spark Driver 进行修改，以减少 speculation 的时间。
- TimSort issue due to integer overflow for large buffer（SPARK-13850）：我们发现 Spark 的 unsafe 内存操作有一个 bug，这会导致 TimSort 中的内存出现问题。不过 Databricks 的工作人员已经修复了这个问题，使我们能够在大型内存缓冲区上进行操作。
- Tune the shuffle service to handle large number of connections：在 shuffle 阶段，我们看到许多 executors 在试图连接 shuffle service 时超时。通过增加 Netty

服务线程（spark.shuffle.io.serverThreads）和 backlog（spark.shuffle.io.backLog）的数量解决了这个问题。

- Fix Spark executor OOM（SPARK-13958）：一开始在每个节点上运行四个以上的 reduce 任务是很有挑战性的。Spark executors 的内存不足，因为 sorter 中存在一个 bug，该 bug 会导致指针数组无限增长。我们通过在指针数组没有更多可用内存时强制将数据溢写到磁盘来修复这个问题。因此，现在我们可以在一个节点上运行24个任务而不会导致内存不足。

性能提升

在实现了上述可靠性改进之后，我们能够可靠地运行 Spark 作业。此时，我们将工作重心转移到与性能相关的问题上，以最大限度地利用 Spark。我们使用Spark 的指标和 profilers 来发现一些性能瓶颈。

我们用来发现性能瓶颈的工具

- Spark UI Metrics：Spark UI 可以很好地洞察特定阶段的时间花在哪里。每个任务的执行时间被划分为子阶段，以便更容易地找到作业中的瓶颈。
- Jstack：Spark UI 中还提供 executor 进程的 jstack 功能，这个可以帮助我们找到代码中的热点问题。
- Spark Linux Perf/Flame Graph support：尽管上面的两个工具非常方便，但它们并没有提供同时运行在数百台机器上作业的 CPU 概要的聚合视图。在每个作业的基础上，我们增加了对性能分析的支持，并且可以定制采样的持续时间/频率。

性能优化

- Fix memory leak in the sorter（SPARK-14363）性能提升 30%：我们发现当任务释放所有内存页，但指针数组没有被释放。结果，大量内存未被使用，导致频繁溢出和 executor OOMs。现在，我们修复了这个问题，这个功能使得 CPU 性能提高了30%；
- Snappy optimization（SPARK-14277）性能提升 10%：对于每一行的读/写，都会调用 JNI 方法（Snappy.ArrayCopy）。我们发现了这个问题，并且将这个调用修改成非 JNI 的System.ArrayCopy调用，修改完之后 CPU 性能提高了10%；
- Reduce shuffle write latency（SPARK-5581）性能提升近 50%：在 map 端，当将 shuffle 数据写入磁盘时，map 任务的每个分区打开和关闭相同的文件。我们修复了这个问题，以避免不必要的打开/关闭，修改完之后 CPU 性能提高近 50%；
- Fix duplicate task run issue due to fetch failure（SPARK-14649）：当获取失败（fetch failure）发生时，Spark driver 会重新提交已经运行的任务，这会导致性能低下。我们通过避免重新运行正在运行的任务修复了这个问题，并且我们发现当发生获取操作失败时，作业也更加稳定。
- Configurable buffer size for PipedRDD（SPARK-14542）性能提升近 10%：在使用 PipedRDD 时，我们发现用于将数据从排序器（sorter）传输到管道处理的默认缓冲区大

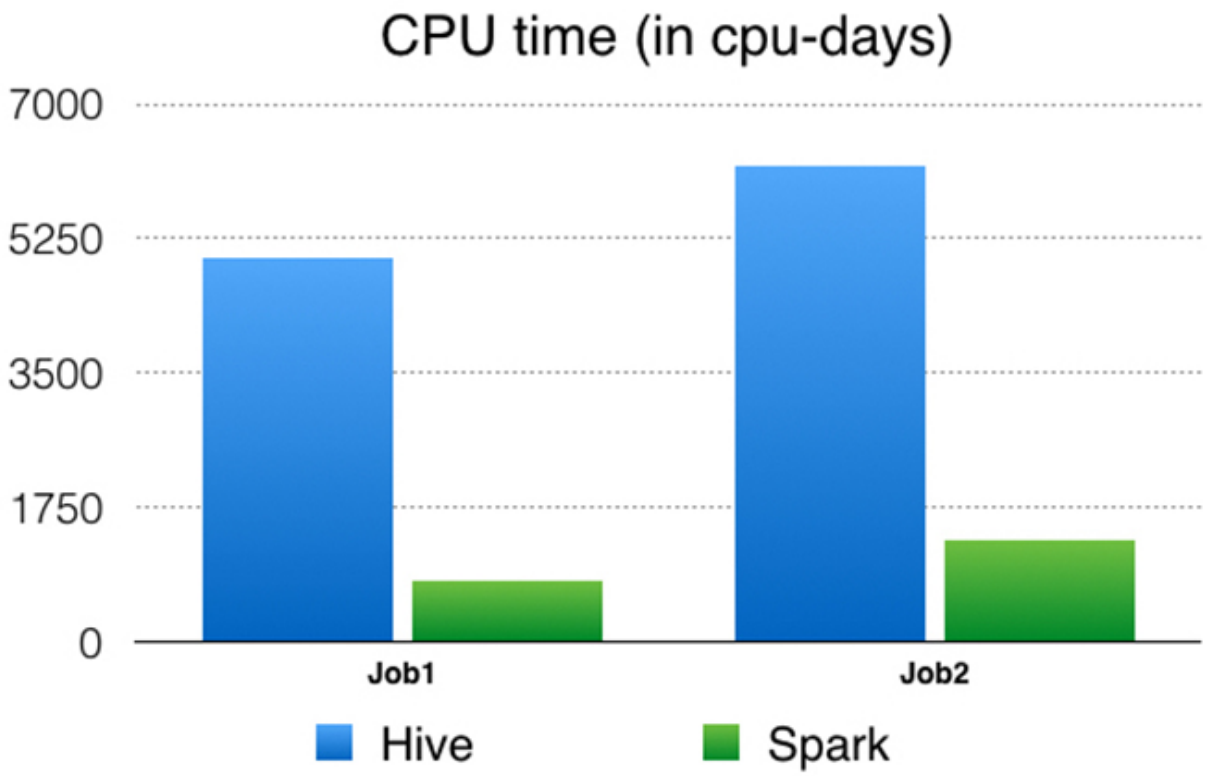
小太小，我们的作业花费了超过 10%
的时间来复制数据。我们使这个缓冲区大小变得可配置，以避免这个瓶颈。
- Cache index files for shuffle fetch speed-up（SPARK-15074）：我们发现，shuffle service 经常成为瓶颈，reduce 端花费 10% 到 15% 的时间来等待获取 map 端的数据。通过更深入的研究这个问题，我们发现 shuffle service 为每次 shuffle fetch 都需要打开/关闭 shuffle index 文件。我们通过缓存索引信息，这样我们就可以避免重复打开/关闭文件，这一变化减少了50%的 shuffle fetch 时间；
- Reduce update frequency of shuffle bytes written metrics（SPARK-15569）性能提升近 20%：使用 Spark Linux Perf 集成，我们发现大约 20% 的 CPU 时间花在探测和更新随机字节写的指标上。
- Configurable initial buffer size for Sorter（ SPARK-15958 ）性能提升近 5%：Sorter 的默认初始缓冲区大小太小(4 KB)，对于大的工作负载来说这个值太小了，因此我们浪费了大量的时间来复制内容。我们将这个缓冲区大小变得可配置（过往记忆大数据备注：spark.shuffle.sort.initialBufferSize），当将这个参数设置为 64 MB 时，可以避免大量的数据复制，使得性能提升近 5%；
- Configuring number of tasks：由于我们输入的数据大小为 60 T，每个 HDFS 块大小为 256 M，因此我们要生成超过250,000个任务。尽管我们能够运行具有如此多任务的 Spark 作业，但我们发现，当任务数量过高时，性能会显著下降。我们引入了一个配置参数，使 map 输入大小可配置，我们通过将输入的 split 大小设置为 2 GB ，使得 task 的数据减少了八倍。

在所有这些可靠性和性能改进之后，我们的实体排名系统变成了一个更快、更易于管理的管道，并且我们提供了在 Spark 中运行其他类似作业的能力。

使用 Spark 和 Hive 运行上面实体排名程序性能比较

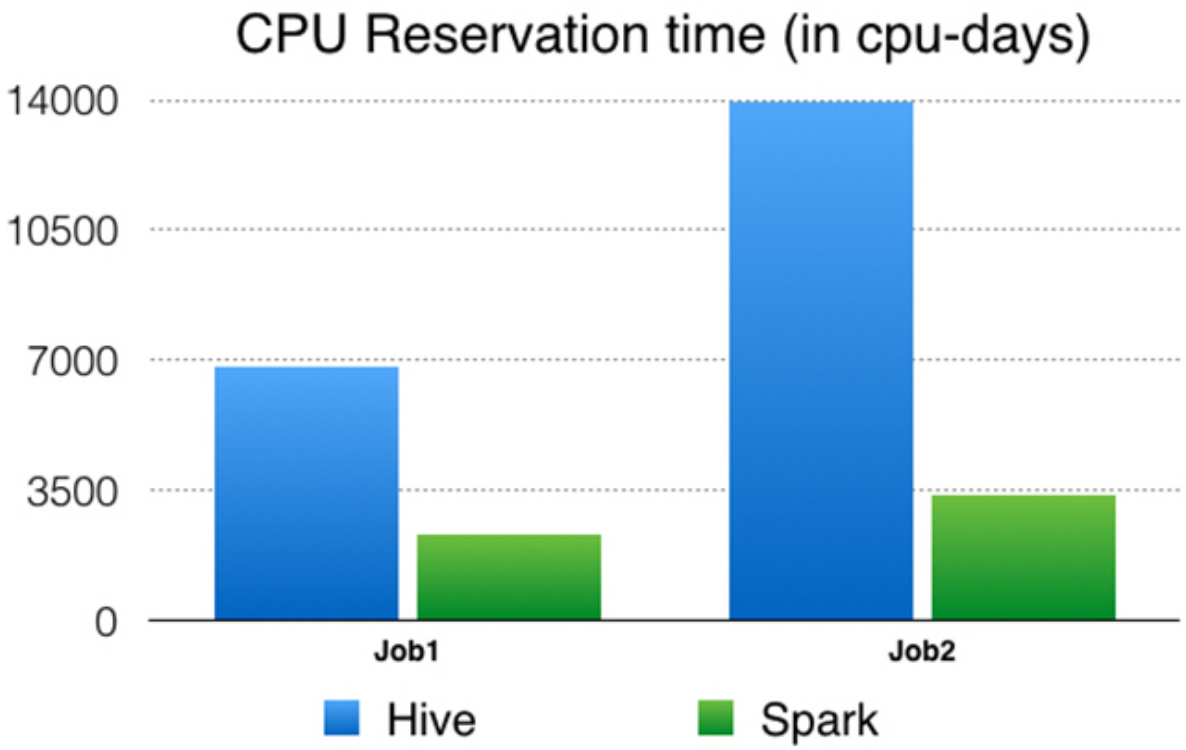
我们使用以下性能指标来比较 Spark 和 Hive 运行性能。

CPU time：这是从操作系统的角度来看 CPU
使用情况。例如，如果您的作业在32核机器上仅运行一个进程，使用所有 CPU
的50%持续10秒，那么您的 CPU 时间将是 32 * 0.5 * 10 = 160 CPU 秒。



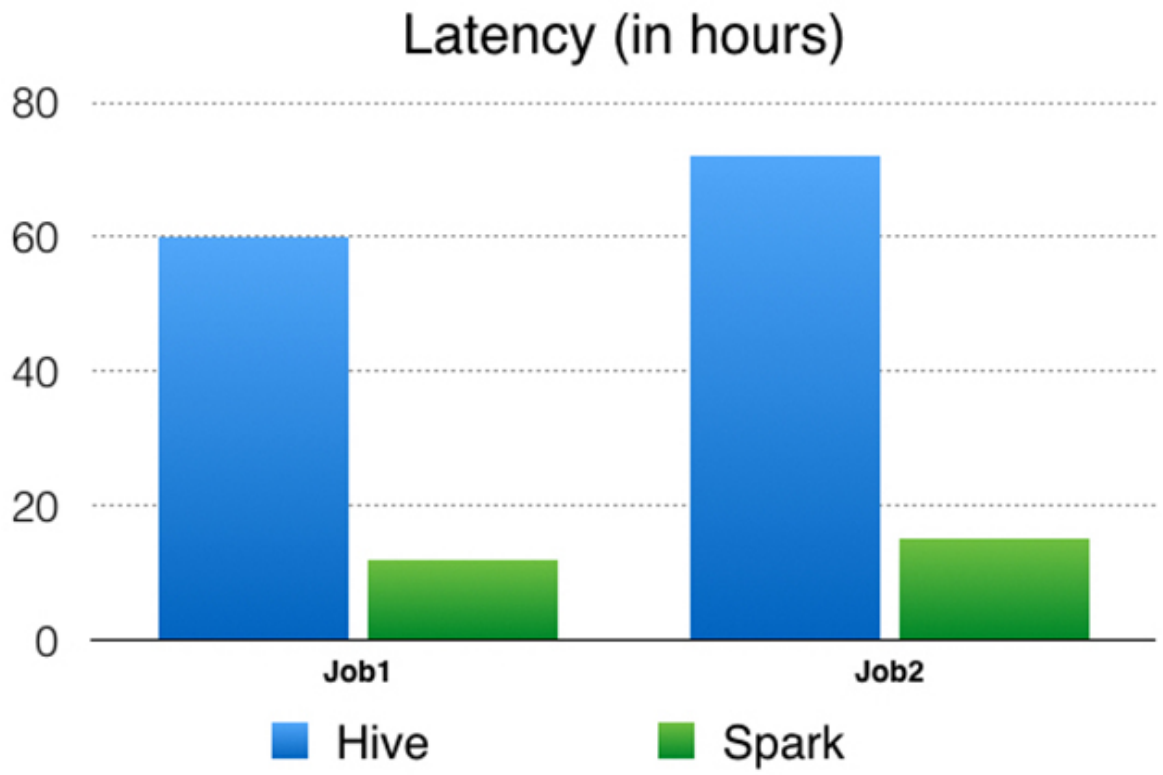
如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

CPU reservation time：从资源管理框架的角度来看，这是 CPU 预留（CPU reservation）。例如，如果我们将32核机器预留10秒来运行这个作业，那么 CPU 预留时间是 32 * 10 = 320 CPU秒。CPU 时间与 CPU 预留时间的比率反映了我们集群预留 CPU 资源的情况。准确地说，当运行相同的工作负载时，与 CPU 时间相比，预留时间可以更好地比较执行引擎。例如，如果一个进程需要1个 CPU 秒来运行，但是必须保留100个 CPU 秒，那么根据这个指标，它的效率低于需要10个 CPU 秒但只预留10个 CPU 秒来做相同数量的工作的进程。我们还计算了内存预留时间，但这里没有列出来，因为这些数字与 CPU 预留时间类似，而且使用 Spark 和 Hive 运行这个程序时都没有在内存中缓存数据。Spark 有能力在内存中缓存数据，但由于集群内存的限制，我们并没有使用这个功能。



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

Latency：作业从开始到结束运行时间。



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

结论和未来工作

Facebook 使用高性能和可扩展的分析引擎来帮助产品开发。Apache Spark 提供了将各种分析用例统一到单个 API ，并且提供了高效的计算引擎。我们将分解成数百个 Hive 作业管道替换为一个 Spark 作业，通过一系列的性能和可靠性改进，我们能够使用 Spark 来处理生产中的实体数据排序的用例。在这个特殊的用例中，我们展示了 Spark 可以可靠地 shuffle 并排序 90 TB 以上的中间数据，并在一个作业中运行 250,000个 tasks。与旧的基于 Hive 计算引擎管道相比，基于 Spark 的管道产生了显著的性能改进（4.5-6倍 CPU性能提升、节省了 3-4 倍资源的使用，并降低了大约5倍的延迟），并且已经在生产环境中运行了几个月。

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: 【】（）