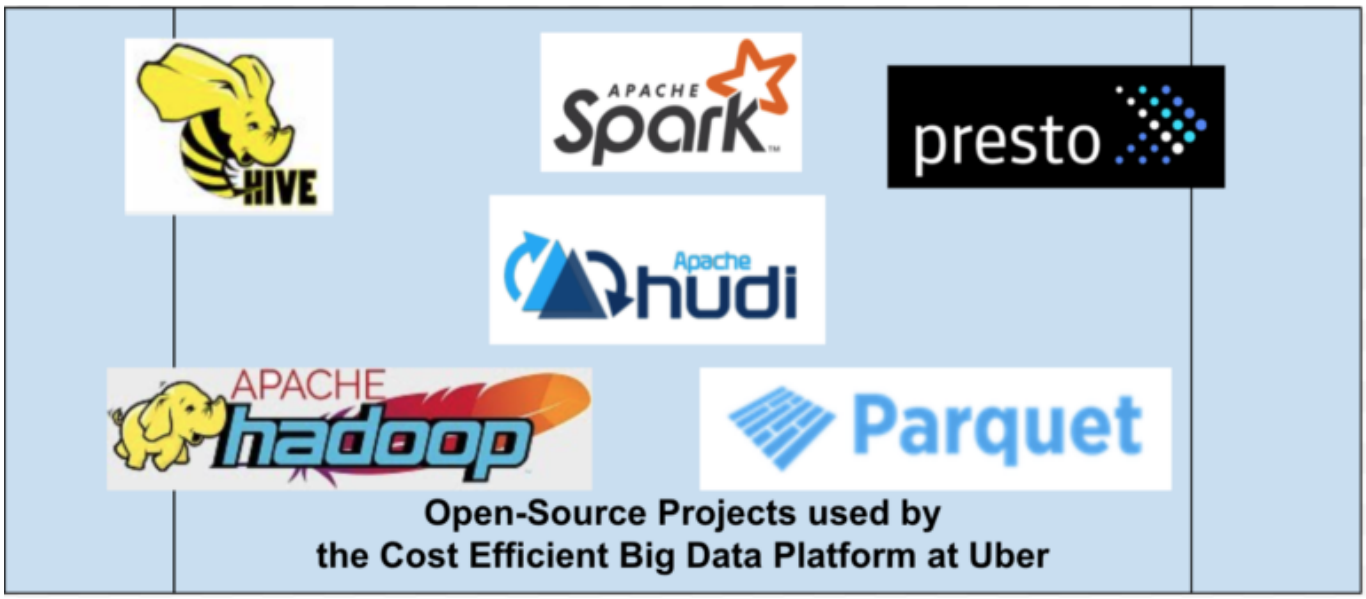
Uber 是如何减少大数据平台的成本

随着 Uber 业务的扩张，为其提供支持的基础数据呈指数级增长，因此处理成本也越来越高。 当大数据成为我们最大的运营开支之一时，我们开始了一项降低数据平台成本的举措，该计划将挑战分为三部分：平台效率、供应和需求。
本文将讨论我们为提高数据平台效率和降低成本所做的努力。



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：过往记忆大数据

大数据文件格式的优化

我们大部分 Apache® Hadoop® 文件系统 (HDFS) 空间都被 Apache Hive 表占用。这些表以 Apache Parquet 文件格式或 Apache ORC 文件格式存储。尽管我们计划在未来的某个时候整合到 Parquet，但由于许多特定要求，包括场景的兼容性和性能，我们还未能实现这一目标。

Parquet 和 ORC 文件格式都是基于块的列格式（block-based columnar formats），这意味着文件包含许多块，每个块包含大量的行（假设为 10,000），这些行数据是拆分成列的数据存储的。

我们花了很多时间研究 HDFS 上的文件，并决定进行以下优化，主要集中在 Parquet 格式：

- 压缩算法：默认情况下，我们使用 GZIP Level 6 作为 Parquet 内部的压缩算法。最近社区关于 Parquet 支持 Facebook 的 ZSTD 压缩算法的 ISSUE 引起了我们的注意。在我们的实验中，与基于 GZIP 的 Parquet 文件相比，ZSTD Level 9 和 Level 19 能够将我们的 Parquet 文件大小分别减少 8% 和 12%。此外，ZSTD 9 级和 19 级的解压速度都比 GZIP 6 级快。我们决定在 1 个月后采用 ZSTD 9 级重新压缩我们的数据，并在 3 个月后采用 ZSTD 级 19

级进行压缩。这是因为在我们的实验中，ZSTD 9 级压缩比 19 级快 3 倍。请注意，重新压缩作业是后台维护作业，可以使用无保证的计算资源运行。鉴于此类资源的丰富性，我们基本上可以将这些再压缩作业视为无开销的。

- 列删除：我们的许多 Hive 表，尤其是从 Apache Kafka 日志中提取的表，都包含许多列，其中一些是嵌套的。当我们查看这些列时，发现其中一些列没有必要长期保留。这些例子包括调试每个 Kafka 消息的元数据以及由于合规性原因需要在一段时间后删除的任何字段。对于列格式文件来说，在技术上可以删除文件内的列，而无需解压和重新压缩其他列。这使得列删除成为一种非常节省 CPU 的操作。我们在 Uber 实现了这样一个功能，在我们的 Hive 表上广泛使用它，并将代码贡献回 Apache Parquet，具体可以参见 #755。
- 重新排序行数据：行顺序可以显着影响 Parquet 文件的压缩大小。这是由于 Parquet 格式使用 Run-Length Encoding 功能，以及压缩算法利用局部重复的能力。我们检查了 Uber 最大的那些 Hive 表，并手动执行了排序，这使表大小减少了 50% 以上。我们发现的一个常见模式是简单地按用户 ID 对行进行排序，然后是日志表的时间戳。大多数日志表都有用户 ID 和时间戳这些列。这使我们能够非常好地压缩与用户 ID 关联的许多非规范化列。
- Delta 编码（Delta Encoding）：一旦我们开始按时间戳对行进行排序，我们很快注意到 Delta 编码可以帮助进一步减少数据大小，因为与时间戳值本身相比，相邻时间戳之间的差异非常小。在某些情况下，日志具有稳定的节奏，就像心跳一样，因此差异是恒定的。但是，我们广泛使用的 Apache Hive、Presto 和 Apache Spark 的环境中，在 Parquet 中启用 Delta 编码并不容易，正如 StackOverflow 这个问题 中所述的。不过我们还在探索这个方向。

HDFS EC

纠删码（Erasure Coding）可以显着降低 HDFS 文件的复制因子。由于潜在增加的 IOPS 工作负载，在 Uber，我们主要研究 3+2 和 6+3 方案，复制因子分别为 1.67 倍和 1.5 倍。鉴于默认的 HDFS 复制因子是 3x，我们可以将 HDD 空间需求减少近一半！

不过，擦除代码有多种选择：

- Apache Hadoop 3.0 HDFS Erasure Code：这是在 Apache Hadoop 3.0 中官方实现的纠删码。这个实现的好处是它既适用于大文件，又适用于小文件。缺点是 IO 效率不高，因为 Erasure Code 的块非常碎片化。
- Client-side Erasure Code：这个 是 Facebook 在 HDFS-RAID 项目中首次实现的。这种方法的好处是它的 IO 效率非常高。当所有块都可用时，读取 IO 效率与三副本的方式基本相同。缺点是它不适用于小文件，因为每个块都是纠删码计算的一个单位。

在咨询了行业专家后，我们决定采用 Apache Hadoop 3.0 HDFS Erasure Code，因为这是社区的方向。我们仍处于 Apache Hadoop 3.0 HDFS 纠删码的评估阶段，但我们相信这将对降低我们的 HDFS 成本产生巨大影响。

YARN 调度策略改进

在 Uber，我们使用 Apache YARN 来运行我们大部分的大数据计算工作负载（Presto 除外，Presto 是直接在专用服务器上运行）。就像许多其他公司一样，我们从 YARN 中的标准容量调度（Capacity Scheduler）开始。Capacity Scheduler 允许我们为每个队列配置具有 MIN 和 MAX 设置的分层队列结构。我们创建了一个以组织为第一级的 2 级队列结构，允许用户根据子团队、优先级或工作类型创建第二级队列。

虽然容量调度为我们管理 YARN 队列容量提供了一个良好的开端，但我们很快就开始面临管理 YARN 集群容量的困境：

- 高利用率：我们希望 YARN 集群的平均利用率（以分配的 CPU 和 Mem GB /集群的总 CPU 和 MemGB 容量衡量）尽可能高；
- 满足用户期望：我们希望让用户清楚地知道他们希望从集群获得多少资源

我们的许多用户对 YARN 集群的资源需求有一些尖锐的但可以预测的需求。例如，一个队列可能有一组日常作业，每个作业都在一天的特定时间开始，并在相同的时间内消耗相同数量的CPU/MemGB。

如果我们将队列的 MIN 设置为白天的峰值使用量，那么集群利用率将非常低，因为队列的平均资源需求远低于 MIN。

如果我们将队列的 MAX 设置为白天的高峰使用，那么随着时间的推移，队列可能会被滥用，不断将资源接近 MAX，进而可能影响其他队列中其他人的正常工作．
我们如何捕捉用户的资源需求并正确设定他们的期望？ 我们提出了以下想法，称为 Dynamic MAX。Dynamic MAX 算法使用以下设置：

- 设置队列的 MIN 为队列的平均使用率；
- 将队列的 MAX 的设置使用以下公式：
  Dynamic_MAX = max(MIN, MIN * 24 – Average_Usage_In_last_23_hours * 23)

Dynamic_MAX 在每小时开始时计算，并应用于该小时的队列 MAX。

这里的 Dynamic MAX 算法背后的直觉是：

- 如果队列在过去 23 小时内根本没有使用，我们允许队列峰值最多达到其 MIN 的 24 倍。这通常足以处理我们绝大多数的峰值工作负载；
- 如果队列在过去 23 小时内平均使用的资源为 MIN，那么我们只允许队列在下一个小时最多使用 MIN 的资源。有了这个规则，队列在 24 小时内的平均使用量不会超过 MIN，从而避免了上面提到的滥用情况。

上面的 Dynamic MAX 算法很容易向用户解释：基本上他们的使用量最多可以飙升到他们队列 MIN 的 24 倍，但是，为了集群的公平性，它们在24小时内的累计使用量不能超过 MIN 水平上的恒定使用量。

实际上，我们将 MIN 设置为队列平均使用量的 125%，以解决高达 25% 的每日使用差异。这反过来意味着我们的 YARN 集群的平均利用率（以 CPU/MemGB 分配衡量）将在 80% 左右，这对于成本效率来说是一个相当不错的利用率水平。

避开高峰时间

YARN 资源利用率的另一个问题是整个集群级别仍然存在定时任务。许多团队决定在 00:00-01:00 UTC 之间运行他们的 ETL 管道，因为据说那是日志准备好的时刻。这些管道可能会运行 1-2 个小时，这使得 YARN 集群在那些高峰时段非常忙碌。

我们计划实施基于时间的速率（time-based rates），而不是向 YARN 集群添加更多机器，这会降低平均利用率并损害成本效率。基本上，当我们计算过去 23 小时的平均使用量时，我们会应用一个根据一天中的小时而不同的比例因子。 例如，0-4 UTC 高峰时段的比例因子为 2 倍，其余时间为 0.8 倍。

集群联邦

随着 YARN 和 HDFS 集群不断变大，我们开始注意到性能瓶颈。由于集群大小不断增加，HDFS NameNode 和 YARN ResourceManager 都开始变慢。
虽然这主要是一个可扩展性挑战，但它也极大地影响了我们的成本效率目标。

为了解决这个问题，摆在我们面前的战略选择有两个：

- A. 继续提升单节点性能：比如我们可以使用更多 CPU vCores 和 Memory 的机器。我们还可以运行堆栈跟踪和火焰图来找出性能瓶颈并逐个进行优化。
- B. 集群联邦：我们可以创建一个由许多集群组成的虚拟集群。每个底层集群的大小都适合 HDFS 和 YARN 的最佳性能。上面的虚拟集群将处理所有工作负载路由逻辑。

我们选择 B 方案的原因如下:

- 世界上大多数 HDFS 和 YARN 集群都比我们在 Uber 需要的要小。如果我们运行超大规模的集群，我们很可能会遇到在较小规模的集群中不会出现的未知bugs。
- 为了使 HDFS 和 YARN 能够扩展到 Uber 的集群规模，我们可能需要更改源代码以在性能和复杂功能之间做出不同的权衡。例如，我们发现容量调度器有一些复杂的逻辑会减慢任务分配的速度。但是，为摆脱这些而进行的代码更改将无法合并到 Apache Hadoop 主干中，因为其他公司可能需要这些复杂的功能。

为了让我们能够在不分叉的情况下利用开源 Hadoop 生态系统，我们决定构建我们的联邦集群。特别是，我们使用基于路由器的 HDFS 联邦和 YARN 联邦。 它们都来自开源 Apache Hadoop。 截至目前，我们已经建立了数十个 HDFS 集群和少数 YARN 集群。HDFS Router-based Federation 一直是我们大数据可扩展性工作的基石，它也提高了成本效率。

通用的负载均衡

在本节中，我们将讨论通用的负载均衡解决方案，它适用于 HDFS 和 YARN，方法如下：

- HDFS DataNode 磁盘空间利用率平衡：每个 DataNode 可能有不同的磁盘空间利用率比率。 在每个 DataNode 中，每个 HDD 可能具有不同的磁盘空间利用率。所有这些都需要平衡，以实现较高的平均磁盘空间利用率。
- YARN NodeManager 利用率平衡：在任何时间点，YARN 中的每台机器都可以有不同级别的 CPU 和 MemGB 分配和利用率。同样，我们需要平衡分配和利用率，以允许较高的平均利用率。

上述解决方案之间的相似性导致了广义负载平衡（generalized load balancing）思想，它适用于我们大数据平台内外的更多用例，例如微服务负载均衡和主存储负载均衡。所有这些之间的共同联系是，目标始终是缩小 P99 与平均值之间的差距。

查询引擎

我们在 Uber 的大数据生态系统中使用了几个查询引擎：Hive-on-Spark、Spark 和 Presto。这些查询引擎与文件格式（Parquet 和 ORC）相结合，为我们的成本效率工作创建了一个有趣的权衡矩阵。 包括 SparkSQL 和 Hive-on-Tez 在内的其他选项使决策变得更加复杂。

以下是我们查询引擎提高成本效率的主要工作点：

- 专注于 Parquet 文件格式：Parquet 和 ORC 文件格式共享一些共同的设计原则，如行组、列存储、块级和文件级统计信息。但是，它们的实现是完全独立的，并且与我们在 Uber 使用的其他专有系统具有不同的兼容性。随着时间的推移，我们发现 Spark 中对 Parquet 文件有更好的支持，在 Presto 中对 ORC 文件有更好的支持。鉴于对文件格式添加功能的需求不断增长，我们必须决定一种主要的文件格式，我们选择了 Parquet。单一的主要文件格式使我们能够将精力集中在一个单一的代码库中，并随着时间的推移积累专业知识。
- 嵌套列裁剪（Nested Column Pruning）：Uber 的大数据表具有惊人的高度嵌套数据。这部分是因为我们的许多上游数据集都以 JSON 格式存储，并且我们在这些数据集上强制使用 Avro 模式。因此，对嵌套列裁剪的支持是 Uber 查询引擎的一个关键特性，否则深度嵌套的数据将需要从 Parquet 文件中完全读出，即使我们只需要嵌套结构中的单个字段 . 我们为 Spark 和 Presto 添加了嵌套列裁剪功能。这些显着提高了我们的整体查询性能，这些功能已经回馈了开源社区。
- 常见查询模式优化：在我们的工作负载中看到接近一千行的 SQL 查询并不少见。虽然我们使用的查询引擎都有一个查询优化器，但它们并没有专门处理 Uber 常见的查询场景。其中一个例子是使用 SQL constructs，如“RANK() OVER PARTITION”和“WHERE rank = 1”，目的是提取另一列值处于最大值的行中某一列的值，或用数学术语来说是 “ARGMAX”。 当查询被重写为使用内置函数“MAX_BY”时，像 Presto 这样的引擎可以运行得更快。

根据我们的经验，很难预测哪个引擎最适合特定的 SQL 查询。 Hive-on-Spark 通常对于大量

shuffle 数据具有很高的可扩展性。 反过来，对于涉及少量数据的查询，Presto 通常非常快。我们正在积极关注开源大数据查询引擎的改进，并将继续在这些查询引擎之间切换我们的工作负载以优化成本效率。

Apache Hudi

我们在大数据平台中最大的成本效益之一是高效的增量处理。我们的许多事实数据集可能会延迟到达或被更改。例如，在许多情况下，乘客直到他或她准备要求下一次行程时才会对上次行程的司机进行评分。信用卡对旅行的退款有时可能需要一个月的时间来处理。

如果没有高效的增量处理框架，我们的大数据用户必须每天扫描许多天的旧数据，才能使他们的查询结果保持新鲜。一种更有效的方法是每天只处理增量更改。 这就是 Hudi 项目的意义所在。

我们在 2016 年启动了 Hudi 项目，并于 2019 年将其提交给 Apache Incubator Project。Apache Hudi 现在是一个 Apache 顶级项目（Top-Level Project），我们在 HDFS 上的大部分大数据都是 Hudi 格式。这大大降低了 Uber 的计算能力需求。

下一步和挑战

大数据和在线服务同主机托管

虽然我们决定让大数据工作负载在在线服务不需要对应主机时使用在线服务的主机，但让两个工作负载在同一主机上运行会带来许多额外的挑战。

关于托管对性能的影响这一领域有很多研究论文。我们方法的主要区别在于，我们计划为大数据工作负载提供非常低的优先级，以尽量减少其对在线服务的影响。

在线和分析存储的融合

我们的很多数据集都存储在在线存储系统（比如 MySQL 数据库中）和分析存储系统（比如 Hive 表）中。此外，为了实现即时查询速度，我们还使用了 Pinot
等存储引擎。所有这些都导致了相同逻辑数据的许多副本，尽管以不同的格式存储。

是否可能有一个统一的存储系统，既能处理在线查询，又能处理分析查询？这将大大降低存储成本。

Project HydroElectricity：利用维护工作来“存储”额外的计算能力

集群中的计算能力与电力供应非常相似。它通常在供应方面是固定的，并且在需求激增或不一致的情况下会受到影响。

抽水蓄能水力发电（Pumped-Storage hydroelectricity）可以将多余的电力以水的重力势能的形式储存起来，然后在需求高峰时将其转换回电力。

我们可以将相同的想法应用于计算能力吗？ 我们可以！

这里要介绍的关键思想是维护作业（maintenance jobs），它们是可以在第二天甚至一周内随时发生的后台任务。典型的维护工作包括 LSM 压缩、compression、二级索引构建、数据清理、纠删码修复和快照维护。几乎所有没有严格 SLA 的低优先级作业都可以视为维护作业。

在我们的大多数系统中，我们没有明确拆分维护和前台工作。例如，我们的 Big Data Ingestion 系统写入使用 ZSTD 压缩的 Parquet 文件中，这会占用大量 CPU 资源并生成非常紧凑的文件。除了这样做之外，我们还可以让 Ingestion 编写轻度压缩的 Parquet 文件，这些文件占用更多磁盘空间但 CPU 更少。 然后我们有一个维护作业，它会晚一点时间运行以重新压缩文件。通过这种方式，我们可以显着减少前台 CPU 的需求。

维护作业可能需要非保证的计算能力才能运行。正如我们之前所描述的，我们有足够的资源用于此目的。

Big Data Usage 的定价机制

考虑到在多租户的大数据平台中，我们经常处于很难满足每个客户的资源需求的情况下。我们如何优化有限的硬件预算的总效用？带有高峰时间乘数的 Dynamic_MAX 是最佳选择吗？

事实上，我们相信还有更好的解决方案。然而，这需要我们想出一个更微妙的定价机制。我们想要考虑的例子包括每个团队可以花在我们集群上的假钱（fake money），用户可以用来提高他们工作优先级的积分等。

总结

在这篇博文中，我们分享了在提高 Uber 大数据平台效率方面的努力和想法，包括文件格式改进、HDFS 纠删码、YARN 调度策略改进、负载均衡、查询引擎和 Apache Hudi。 这些改进带来了显着的成本减少。 此外，我们探索了一些开放性的挑战，例如分析和在线托管以及定价机制。然而，正如我们之前文章中概述的框架所建立的那样，平台效率的提升并不能保证高效的运行。控制数据的供应和需求同样重要，我们将在下一篇文章中讨论。

本文翻译自：Cost-Efficient Open Source Big Data Platform at Uber

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: 【】（）