Magnet: 基于推送的大规模数据处理 Shuffle 服务

本文翻译自：《Magnet: Push-based Shuf□e Service for Large-scale Data Processing》

在过去的十年中，Apache Spark 已成为大规模数据处理的流行计算引擎。与其他基于 MapReduce 计算范式的计算引擎一样，随机Shuffle操作（即中间数据的全部对全部传输）在 Spark 中起着重要作用。在 LinkedIn，随着数据量和 Spark 部署规模的快速增长，随机Shuffle操作正成为进一步扩展基础设施的瓶颈。这导致整体作业变慢，甚至长时间运行的作业失败。这不仅影响开发者解决此类缓慢和故障的生产力，还导致基础设施的高运营成本。

在这项工作中，我们描述了影响随机Shuffle可扩展性的主要瓶颈。我们提出了一种新颖的随机Shuffle机制——磁石，它可以扩展以处理每天成千上万 PB 的随机Shuffle数据和拥有数千节点的集群。磁石设计用于本地和云端集群部署。它通过将分散的中间随机Shuffle数据合并成大块来解决关键的随机Shuffle可扩展性瓶颈。磁石通过将合并块与 Reduce Task 放置在一起，提供了进一步的改进。我们的基准测试显示，磁石显著提高了随机Shuffle性能，独立于底层硬件。磁石使 LinkedIn 生产 Spark 作业的端到端运行时间减少了近 30%。此外，Magnet 通过消除与数据Shuffle相关的调整负担，提高了用户的生产力。

分布式数据处理框架，如 Hadoop[1]和 Spark[40]，在过去十年因大规模数据分析用例而变得流行。基于此 MapReduce 计算范式[22]以及利用一整套商品机器，这些分布式数据处理框架展现了良好的可伸缩性和广泛的适用性，适用于从数据分析到机器学习和人工智能等多种用例。近年来，一系列现代计算引擎如 Spark SQL[18]、Presto[35]和 Flink[19]涌现并走向主流。与 Hadoop MapReduce 不同，这些现代计算引擎利用 SQL 优化器来优化用户指定的计算逻辑，然后将其交给 DAG 执行引擎执行优化后的操作。以 Spark 为例（见图 1）。假设用户希望在基于某些条件过滤连接结果之前，对 job_post_view 表和 job_dimension 表执行内连接。在这个例子中，前者包含 LinkedIn 平台上哪个成员查看了哪个职位帖子的追踪信息，后者包含每个职位帖子的详细信息。Spark

通过在连接操作之前下推过滤条件来优化这个查询（见图 1(a)）。然后，Spark 的 DAG 执行引擎将这个优化后的计算计划转换为一个或多个作业。每个作业由一个阶段的 DAG 组成，表示数据如何被转换以产生当前作业的最终结果（见图 1(b)）。阶段之间的中间数据通过Shuffle操作进行传输。

Shuffle操作是关键，它通过 map 任务和缩减任务之间所有到所有的连接传输中间数据，对应于 MapReduce 计算范式[22]。尽管Shuffle操作的基本概念很直接，但不同的框架采取了不同的方法来实现它。一些框架，如 Presto[35]和 Flink 流处理[19]，为了满足低延迟需求，将中间Shuffle数据实现在内存中；而其他框架，如 Spark[40]和 Flink 批处理[19]，则将其实现在本地磁盘上，以获得更好的容错性。当在磁盘上实现中间Shuffle数据时，有基于哈希的解决方案，其中每个 map 任务为每个缩减任务生成单独的文件；还有基于排序的解决方案，其中 map 任务的输出按分区键的哈希值进行排序，并作为一个单独的文件实现。虽然基于排序的Shuffle会产生排序的开销，但当中间Shuffle数据量大时，它是一种更具性能和可扩展性的解决方案[21,32]。像 Spark 和 Flink 这样的框架已经采用了外部Shuffle服务[5, 7, 11]来提供已实现的中间Shuffle数据，以实现更好的容错性和性能隔离。随着最近在网络和存储硬件中的改进，一些解决方案[36, 15]将中间Shuffle数据以分散存储的方式实现，而不是本地存储。其他解决方案[20, 23]则绕过了中间Shuffle数据的实现，通过直接将 map 任务的输出推送到 Reduce Task ，以实现低延迟。这种多样的Shuffle实现为这些计算引擎中的Shuffle操作提供了丰富的优化空间。实际上，提升 Spark 中的Shuffle效率是其赢得排序基准测试[2]的关键。