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。

在 LinkedIn
这个全球专业的社交网络公司，每天有非常大量的批量分析和机器学习作业在我们的生产 Spark 集群上运行，这些集群跨越了成千上万个节点。这导致 Spark 每天需要处理 PB 级别的数据Shuffle。在处理如此巨量数据时，Shuffle操作对 Spark 基础设施的运行至关重要。Spark 以基于排序的方式将Shuffle数据实现在磁盘上，并通过外部Shuffle服务提供。虽然这种方式提供了良好的容错和性能平衡，但 LinkedIn 的 Spark 工作负载的快速增长仍然带来了多重挑战。

首先，建立全连接以传输 map 任务和 Reduce Task 之间的Shuffle数据的要求带来了可靠性问题。在拥有数千节点的集群中，个别节点间歇性不可用的情况可能很常见。在高峰时段，增加的Shuffle工作负载也可能给部署的Shuffle服务带来压力，进一步增加连接失败的可能性。

其次，Shuffle过程中产生的磁盘 I/O 操作存在效率问题。物化的 Spark Shuffle数据被分割成Shuffle块，这些块以随机顺序单独获取。这些块通常很小。LinkedIn Spark 集群中的平均块大小只有大约几十 KB。我们的集群每天要读取数十亿这样的块，如果从 HDD（硬盘驱动器）中提供服务，可能会严重地给磁盘带来压力。小的随机磁盘读取，加上其他开销如小网络 I/O，导致获取Shuffle数据的延迟增加。由于这种延迟，我们集群上总 Spark 计算资源的约 15%被浪费。

最后，正如[41]中指出的那样，随着Shuffle数据量的增长，平均块大小的减小也引入了一个可扩展性问题。随着我们的 Spark 工作负载趋向于处理更多数据，这个效率问题逐渐恶化。一些配置不当、不必要的块过小的 Spark 应用程序进一步加剧了这个问题。

虽然像将Shuffle数据存储在 SSD（固态硬盘）上这样的解决方案可以帮助缓解这些问题，但在 LinkedIn 规模上用 SSD 替换 HDD 并不实际。有关这方面的更多细节在第 2.2 节中讨论。此外，利用非聚合存储的基于云的集群部署也由于网络开销而遭受小读取的影响。为了应对这些挑战，我们提出了一种名为“Magnet”的替代Shuffle机制。通过Magnet，我们将每个地图任务产生的碎片化Shuffle块推送到远程Shuffle服务，并有机会将它们合并成每个Shuffle分区的大块。其部分优势如下：

 Magnet机会性地将碎片化的中间Shuffle数据合并成大块，并与 Reduce Task 共置。这使得Magn

et能
够显著提
高Shuffle操作的效
率，并减少作业的端到端运行时间，
与底层存储硬件无关。 Magnet采用混合方法，合并和未合并的Shuffle数据都可以作为 Reduce Task
的输入。这有助于提高Shuffle过程中的可靠性。
Magnet设计适用于本地或基于云的部署，并且可以扩展以处理每天Shuffle数据的拍字节级和拥有数千节点的集群。

本文的其余部分组织如下：第 2 节介绍 Spark Shuffle操作并讨论现有问题。第 3 节详细介绍了Magnet的设计。第 4 节展示了在实现Magnet过程中采用的一些优化。第 5 节提供了评估设置、关键结果和分析。第 6 节讨论了相关工作。我们在第 7 节总结本文。

在本节中，我们将为Magnet提供动机并介绍其背景。第 2.1 节回顾了当前的 Spark Shuffle操作。第 2.2-2.4 讨论我们在超大规模运营 Spark 基础设施时遇到的主要Shuffle问题。

2.1 当前的 Spark Shuffle 操作

如前所述，阶段之间的中间数据通过Shuffle操作进行传输。在 Spark 中，Shuffle的执行方式会根据部署模式略有不同。在 LinkedIn，我们将 Spark 部署在 YARN 上[37]，并利用外部Shuffle服务[5]来管理Shuffle数据。这种部署模式在业界也被广泛采用，包括一些拥有最大规模 Spark 部署的公司，如 Netflix[10]、Uber[9]和 Facebook[16]。在这种 Spark 部署下，Spark 中的Shuffle操作如图 2 所示：

每个 Spark 执行器在启动时都会向位于同一节点的 Spark 外部Shuffle服务（ESS）注册。此类注册使 Spark ESS 能够知晓由每个已注册执行器的本地 map 任务生成的已物化Shuffle数据的位置。请注意，Spark ESS 实例位于 Spark 执行器的外部，并可能被多个 Spark 应用程序共享。

在 Shuffle map 阶段内的每个任务处理其数据的一部分。在 map 任务结束时，它生成一对文件，

一个用于Shuffle数据，另一个用于索引前者中的Shuffle块。为此，map
任务根据分区键的哈希值对所有转换后的记录进行排序。在此过程中，如果 map 任务无法在内存中对全部数据进行排序，它可能会将中间数据溢出到磁盘。一旦排序完成，就会生成Shuffle数据文件，所有属于同一Shuffle分区的记录被分组放入一个Shuffle块中。同时还会生成匹配的Shuffle索引文件，该文件记录了块边界偏移量。

当下一阶段减少任务开始运行时，它们会向 Spark
驱动器查询其输入Shuffle块的位置。一旦获得这些信息，每个减少任务都会与相应的 Spark ESS 实例建立连接，以获取其输入数据。Spark ESS 在接收到此类请求后，利用Shuffle索引文件跳至Shuffle数据文件中相应的块数据，从磁盘读取后发送回减少任务。通过以基于排序的方式将Shuffle数据实现到磁盘上，Spark Shuffle操作在性能和容错性之间实现了合理的平衡。此外，通过与 Spark 执行器解耦，Spark ESS 为 Spark Shuffle操作带来了额外的好处：1）即使 Spark
执行器正在经历垃圾收集暂停，Spark ESS 也能提供Shuffle块服务。2）即使生成它们的 Spark
执行器已不存在，Shuffle块仍然可以被提供服务。3）空闲的 Spark
执行器可以被释放出来以节省集群计算资源。然而，在 LinkedIn 规模下运营 Spark 基础设施时，我们仍然观察到多个与Shuffle相关的问题，这些问题正成为基础设施可靠性、效率和可扩展性的潜在瓶颈。我们将在以下小节中讨论这些问题。

2.2 Shuffle过程中磁盘 I/O 效率低下

我们在使用 Spark Shuffle操作时观察到的最大问题之一是由于小Shuffle块导致的磁盘 I/O
效率低下。由于 Spark ESS 每次获取请求只读取一个Shuffle块，因此Shuffle块的平均大小决定了每次磁盘读取的平均数据量。在 LinkedIn，我们的 Spark 集群中主要使用 HDD
来存储中间Shuffle数据。这是由于 HDD 在大规模存储需求方面与 SSD 相比具有成本效益[28, 24]，以及 SSD 用于临时数据存储（如Shuffle数据）时容易磨损[29]。对于 HDD

来说，处理大量小随机读取会受限于其 IOPS（每秒 I/O 操作次数）限制。此外，随机打乱数据通常只读取一次，且各个随机打乱块以随机顺序访问。这种数据访问模式意味着缓存无法帮助提高磁盘 I/O 效率。硬盘有限的 IOPS
与随机打乱数据的访问模式的结合导致磁盘吞吐量低，并且获取随机打乱块的延迟延长。

图 3：对 5000 个采样阶段进行随机打乱并减少的散点图，这些阶段存在显著的延迟。图表显示了每个任务的平均随机打乱获取延迟和平均随机打乱块大小。散点图右侧的条形图显示了该数据集中随机打乱块大小的分布情况。大多数数据点的块大小较小。

在 LinkedIn 的生产 Spark 集群中，尽管平均每日随机打乱的数据量很大，到 2019 年底达到每天几拍字节，但平均每日随机打乱块的数量也非常高（数百十亿）。平均随机打乱块大小仅为几十 KB，这导致随机打乱数据的获取延迟。图 3 绘制了平均随机打乱块大小与 5000
个存在显著延迟（每个任务超过 30
秒）的随机打乱减少阶段的平均随机打乱获取延迟之间的相关性。这些阶段是从 2019 年 4
月在我们集群中运行的生产 Spark 作业中抽样的。图表进一步展示了这 5000 个阶段的块大小分布。从这个图表中，我们可以观察到，大多数存在显著随机打乱获取延迟的随机打乱减少阶段都是块大小较小的阶段。

正如[41]中指出的，对于一个由 M 个 Mapper 和 R 个 Reducer 组成的Shuffle操作，如果每个任务处理的数据量保持不变，那么Shuffle块的数量（M*R）将以平方级别增长。

随着随机Shuffle数据 D 的增长，每个任务处理的数据量保持相对恒定是一种经过验证的做法，以避免在工作负载增加时出现困难。横向扩展 Spark 应用程序（更多的执行器）而不是纵向扩展（更大的执行器），可以避免在多租户集群中获取执行器容器的难度增加。这也可以防止由于资源过度配置而导致的资源利用率下降。在
LinkedIn，由于加入平台的会员数量不断增加，以及构建更复杂的 AI 模型的需求，数据分析和机器学习工作负载处理的数据比以往更多。根据上述分析，处理更多数据的这种需求将导致随机Shuffle块大小的必然减小。

2.3 随机Shuffle全连接的可靠性

我们观察到的另一个常见问题是，在随机Shuffle过程中与 Spark ESS 建立 RPC
连接的可靠性。如前所述，在随机Shuffle的 Reduce
阶段，每个任务需要与其任务输入托管的所有远程随机Shuffle服务建立连接。对于一个由 M 个 mapper 和 R 个 reducer 组成的随机Shuffle，理论上需要建立总共 MR
个连接。在实践中，同一执行器上的 reducer 共享一个到目的地的输出连接，而注册到同一个 Spark ESS 的 mapper 则每个源共享一个输入连接。因此，对于一个使用 S 个 Spark
Shuffle服务和 E 个执行器的 Spark 应用程序，仍然需要多达 SE 个连接。在 Spark
基础设施的大规模部署中，S 和 E 都可以达到 1000。对于大规模 Spark
集群，可能会出现间歇性的节点可用性问题。此外，由于 Spark ESS 是一项共享服务，当一个 Spark ESS 受到配置不当的作业影响或在高峰时段接收增加的Shuffle工作量时，它也可能承受压力并降低可用性。当 reduce 任务在建立与远程 Spark ESS
的连接时出现故障，它会立即导致整个Shuffle reduce 阶段失败，从而导致之前阶段的重试以重新生成无法获取的Shuffle数据。这样的重试可能代价非常高昂，并且已经导致 Spark
应用程序运行时间延迟，进而导致生产流程 SLA 中断。

2.4 减少任务的分配

一项研究[17]声称，在数据中心计算中，数据局部性不再重要，而其他研究[30, 26,
38]仍显示出数据局部性所能提供的好处。尽管过去十年网络速度显著提高，但由于第 2.2
节讨论的主轴磁盘 IOPS 的限制，我们往往无法使网络带宽达到饱和。我们在第 5.2
节进行的基准测试中验证了这一点。此外，如果本地有可用的Shuffle块，reduce
任务可以直接从磁盘读取，绕过Shuffle服务。这也有助于减少Shuffle过程中的 RPC
连接数。尽管Shuffle数据局部性可以提供这些好处，但当前 Spark 中的Shuffle机制会导致 reduce 任务的数据局部性较差，因为它们的任务输入数据分散在所有 map 任务中。

针对第 2 节中描述的问题，我们提出了 Mag-net，一种用于 Spark 的替代Shuffle机制。Magnet 旨在保持当前 Spark
Shuffle操作的容错优势，以基于排序的方式实现中间Shuffle数据的物化，同时克服上述问题。

在设计 Magnet 时，我们需要克服几个挑战。首先，Magnet 需要提高Shuffle操作期间的磁盘 I/O 效率。它应该避免从磁盘读取单个小的Shuffle块，这会降低磁盘吞吐量。其次，Magnet
应帮助减轻潜在的 Spark ESS 连接故障，以提高大规模 Spark
集群中Shuffle操作的整体可靠性。第三，Magnet 需要应对可能出现的拖后腿和数据偏斜问题，这些问题在实际生产工作负载的大规模集群中可能很常见。最后，Magnet
需要在不会产生过多内存或 CPU 开销的情况下实现这些好处。这对于使 Magnet
成为一个可扩展的解决方案至关重要，以应对拥有数千个节点和每天数百 PB
Shuffle数据的集群。

3.1 解决方案概述

图 4 展示了 Magnet 的架构。我们对 Spark 中的四个现有组件（Spark Driver 、Spark ESS、Spark Shuffle map 任务和 Spark Shuffle Reduce Task
）进行了扩展，增加了额外的行为。更多细节将在后续小节中介绍。

 Spark Driver 协调 map 任务、 Reduce Task 和Shuffle服务之间的整个Shuffle操作。（图 4 中的步骤 1、6、7） Shuffle map
任务现在处理额外的准备，以便将它们物化的Shuffle数据推送到远程Shuffle服务。（图 4 中的步骤 2、3） Magnet Shuffle服务是一种增强的 Spark ESS，它接受远程推送的Shuffle块并将它们合并到每个唯一Shuffle分区的相应合并Shuffle文件中。（图 4 中的步骤 4） Shuffle Reduce Task 现在利用这些合并的Shuffle文件，这些文件通常与任务本身位于同一位置，以提高获取任务输入的效率。（图 4 中的步骤 8）

Magnet 的一些关键特性在下面简要描述。

Push-Merge Shuffle - Magnet 采用了一种推送-合并Shuffle机制，其中 mapper
生成的Shuffle数据被推送到远程 Magnet Shuffle服务，按Shuffle分区进行合并。这使得 Magnet

能够将小Shuffle块的随机读取转换为 MB 大小块的顺序读取。此外，这个推送操作与 mapper 解耦，因此如果操作失败，它不会增加 map 任务的运行时间或导致 map 任务失败。

Best-effort Approach - Magnet不需要完美完成块推送操作。通过执行推送合并Shuffle，Magnet有效地复制了Shuffle数据。Magnet允许 reducer 获取已合并和未合并的Shuffle数据作为任务输入。这使得Magnet能够容忍块推送操作的部分完成。

灵活的部署策略 - Magnet通过在其上构建来与 Spark 原生Shuffle集成。这使得Magnet既能在计算和存储节点共址的本地集群中部署，也能在具有分散存储层的云集群中部署。在前一种情况下，由于每个 reducer 任务的输入大多数在一个位置合并，Magnet利用这种局部性来调度 reducer 任务，从而实现更好的 reducer 数据局部性。在后一种情况下，Magnet可以通过选择负载较少的远程Shuffle服务来优化负载平衡，而不是依赖数据局部性。

处理掉队者/数据偏斜 - Magnet能处理掉队者和数据偏斜问题。由于Magnet能容忍块推送操作的部分完成，它可以通过停止缓慢的推送操作或跳过推送大型/偏斜的块来减轻掉队者和数据偏斜的影响。

3.2 推送合并Shuffle

为了提高磁盘 I/O 效率，我们需要增加每次 I/O 操作读取的数据量，或者切换到更适用于小随机读写的存储介质，如 SSD 或 PMEM。然而，如第 2 节所述。2. 由于“调优困境”，应用参数调整无法有效增加Shuffle块大小，而 SSD 或 PMEM 存储大规模中间Shuffle数据成本过高。[33, 41] 提出了解决方案他合并属于同一Shuffle分区的Shuffle块，以创建更大的块。这些技术有效地提高了Shuffle过程中的磁盘 I/O 效率。Magnet 采用了一种推送合并Shuffle机制，该机制也利用了Shuffle块合并技术。与[33, 41]相比，Magnet 实现了更好的容错性、减少了数据局部性并提高了资源效率。有关此比较的更多细节在第 6 节中给出。

3.2.1 准备远程推送的块

我们在设计 Magnet 中的块推送操作时遵循的一个原则是尽量减少对Shuffle服务的 CPU/内存开销。如前所述，Spark ESS 是集群中所有 Spark 应用程序的共享服务。对于在 YARN 上部署的 Spark，分配给 Spark ESS 的资源通常比可用于 Spark

执行器容器的资源少几个数量级。在Shuffle服务上产生沉重的
CPU/内存开销会影响其可扩展性。在[41]中，Spark ESS 需要将多个 MB
大小的数据块从本地Shuffle文件缓冲到内存中，以便高效地合并块。在[33]中， mapper 生成的记录没有在本地实现，而是发送到远程Shuffle服务的每个Shuffle分区的写前缓冲区。这些缓冲的记录可能在Shuffle服务上排序，然后才被实现到外部分布式文件中系统。这两种方法都不理想，无法确保Shuffle服务中的低 CPU/内存开销。

在 Magnet 中，我们保留了当前的Shuffle数据实现方式，该方式以排序的方式实现Shuffle数据的实体化。一旦 map 任务生成了Shuffle文件，它会将Shuffle数据文件中的块划分为 MB
大小的块，每个块将被推送到远程的 Magnet Shuffle 服务进行合并。使用 Magnet 时，Spark
Driver 确定了一组给定Shuffle的 map 任务要使用的 Magnet
Shuffle服务与其配合使用。利用这些信息，每个 map
任务可以一致地决定从Shuffle块到块的映射，以及从块到 Magnet
Shuffle服务的映射。更多细节在算法 1 中有描述。

该算法保证每个块只包含Shuffle文件内至一定大小的连续块，并且属于同一Shuffle分区的不同

mapper 的块会被推送到同一个 Magnet Shuffle服务。为了减少同一Shuffle分区中来自不同 map Task 的块同时被推送到同一个 Magnet Shuffle 服务的机会，每个 map Task 随机化处理块的顺序。一旦块被划分并随机化， map Task 就会转交给一个专用线程池来处理这些块的传输，然后结束。每个块从磁盘加载到内存中，其中的单个块被推送到关联的 Magnet Shuffle服务。注意，这种块的缓冲发生在 Spark 执行器内部，而不是 Magnet Shuffle服务。有关此操作的更多细节在第 4.1 节给出。

3.2.2 在 Magnet Shuffle服务上合并块

在 Magnet Shuffle服务端，对于每个正在被合并的活跃Shuffle分区，Magnet Shuffle服务生成一个合并的Shuffle文件，用以追加所有收到的相应块。它还维护每个活跃合并的Shuffle分区的一些元数据。元数据包含一个位图，用于跟踪所有合并的Shuffle块的 mapper ID；一个位置偏移量，表示在最近成功追加的Shuffle块之后，在合并的Shuffle文件中的偏移；以及一个 currentMapId，用于跟踪当前正在追加的Shuffle块的 mapper ID。该元数据由应用 ID、Shuffle ID 和Shuffle分区 ID 的组合唯一标识，并作为 ConcurrentHashMap 维护。如图 5 所示。

当 Magnet Shuffle服务接收到一个块时，它会首先检索相应的Shuffle分区元数据，然后再尝试将该块追加到相应的合并Shuffle文件中。该元数据可以帮助 Magnet Shuffle服务正确处理几种潜在的异常情况。位图帮助 Magnet Shuffle服务识别任何潜在的重复块，因此不会将冗余数据写入合并的Shuffle文件。此外，尽管 map Task 已经随机化了数据块顺序，但 Magnet Shuffle服务可能仍然从不同 map Task 接收到属于同一Shuffle分区的多个块。当这种情况发生时，currentMapId 元数据可以保证在下一个块写入磁盘之前，有一个块被完全追加到合并的Shuffle文件中。此外，在遇到故障之前，一个块可能部分地被追加到合并的Shuffle文件中，这会破坏整个合并的Shuffle文件。当这种情况发生时，位置偏移量可以帮助恢复合并的Shuffle文件到健康状态。下一个数据块将从位置偏移处追加，有效地覆盖损坏部分。如果损坏的数据块是最后一个数据块，那么在数据块合并操作完成时，损坏部分将被截断。通过跟踪这些元数据，Magnet Shuffle服务可以正确处理数据块合并操作期间的重复、冲突和失败情况。

3.3 提高Shuffle可靠性

Magnet 采取尽力而为的方法，并且可以回退到获取原始的未合并Shuffle数据块。因此，在数据块推送/合并操作期间的任何失败都不是致命的：

 如果在 map Task 实现其Shuffle数据之前失败，此时不会触发远程数据块推送。map Task 的正常重试将启动。 如果一个Shuffle数据块未能成功推送到远程的 Magnet Shuffle服务，经过若干次重试后，Magnet

 如果在数据块合并操作期间，Magnet Shuffle服务遇到任何重复、冲突或失败，导致数据块未能合并，则会取而代之获取原始的未合并数据
块。
如果缩减任务未能成功获取已合并的Shuffle数据块，它可以回退到获取支持该已合并Shuffle数据块的未合并Shuffle数据块列表，而不会引发Shuffle获取失败。

如第 3.2.2 节所述，Magnet Shuffle服务中跟踪的元数据主要提高了对合并失败的容忍度。在 Spark Driver 中跟踪的额外元数据有助于提高 reduce 任务的容错能力。如图 4 所示，在第 6 步中，当 Spark Driver 通知它们停止合并给定 shuffle 的块时，它会从 Magnet Shuffle服务检索 MergeStatus 列表。此外，每个 map 任务完成时，它还会向 Spark Driver 报告 MapStatus（见图 4 的第 5 步）。对于一个包含 M 个 map 任务和 R 个 reduce 任务的 shuffle，Spark Driver 会收集 M 个 MapStatus 和 R 个 MergeStatus。这些元数据告诉 Spark Driver 每个未合并的 shuffle 块和已合并的 shuffle
文件的位置和大小，以及哪些块被合并到每个已合并的 shuffle 文件中。因此，Spark Driver 可以构建一个完整的图景，了解如何结合已合并的 shuffle 文件和未合并的块来获取每个 reduce 任务的输入。当一个 reduce 任务未能获取已合并的 shuffle
块时，该元数据还使其能够回退到获取原始的未合并块。

Magnet 的尽力而为方法有效地维护了 shuffle 数据的两个副本。虽然这有助于提高 shuffle 的可靠性，但它也增加了 shuffle
数据的存储需求和写入操作。实际上，前者并不是一个主要问题。shuffle
数据只是临时存储在磁盘上。一旦 Spark 应用程序完成，其所有的 shuffle
数据也会被删除。尽管我们的集群中的 Spark 应用程序每天要处理 PB 级别的数据，但 shuffle 数据的峰值存储需求只有数百
TB。增加的Shuffle数据存储需求仅占我们集群总容量的一小部分。我们在第 4.3
节进一步讨论了增加写操作的影响。

3.4 灵活部署策略

在第 3.2-3.3 节中，我们已经展示了 Magnet 如何通过在其上构建来与 Spark 原生Shuffle集成。与[4, 33]不同，Magnet 允许 Spark 原生管理Shuffle的各个方面，包括存储Shuffle数据、提供容错以及跟踪Shuffle数据位置元数据。在这种情况下，Spark 不依赖外部系统进行Shuffle。这提供了灵活性，可以将 Magnet 部署在计算/存储节点共置的本地集群以及具有分离存储层的基于云的集群中。

对于共置计算和存储节点的本地数据中心，Shuffle减少任务的就近数据访问可以带来多重好处。这包括提高 I/O 效率，以及由于绕过网络传输而减少Shuffle获取失败。通过利用 Spark 的知晓位置的任务调度[12]并根据 Spark 执行器的位置选择 Magnet Shuffle服务来推送Shuffle块，实现Shuffle数据就近看似轻而易举。然而，Spark 的动态资源分配[13]使情况变得复杂。动态分配功能允许 Spark 在一段时间内没有运行任何任务的空闲执行器释放，并在任务再次待处理时稍后重新启动执行器。这使得 Spark 应用程序在多租户集群中更加资源高效。LinkedIn 的 Spark 部署启用了此功能。同样，一些其他大规模使用 Apache Spark 的部署也推荐这样做[10, 16]。

在使用 Spark 动态分配时，当 Driver 在 shuffle map 阶段 开始时选择一组 Magnet Shuffle服务（图 4 中的步骤 1），由于前一个阶段的执行器释放，活跃的 Spark 执行器的数量可能会少于所需的数量。如果我们基于 Spark 执行器的位置来选择 Magnet Shuffle服务，我们最终可能得到的Shuffle服务数量少于所需。为了解决这个问题，我们选择在活跃的 Spark 执行器之外的位置的 Magnet Shuffle服务，并通过基于所选 Magnet Shuffle服务位置的动态分配稍后启动 Spark 执行器。这样，我们不是基于 Spark 执行器的位置来选择 Magnet Shuffle服务，而是基于 Magnet Shuffle服务的位置来启动 Spark 执行器。这种优化之所以可行，是因为 Magnet 与 Spark 原生Shuffle功能的集成。

对于基于云的集群部署，计算和存储节点通常是分离的。在这种部署中，中间Shuffle数据可以通过快速网络连接在分离的存储节点上实现[36,15]。在这种设置下，对于 shuffle Reduce Task 的“数据局部性”不再重要。然而，Magnet 仍然非常适合这种基于云的部署。Magnet Shuffle服务运行在计算节点上，并将合并后的Shuffle文件存储在分离的存储节点上。通过在网络中读取较大的数据块而不是通过小碎片化的Shuffle块进行读取，Magnet 有助于更好地利用可用的网络带宽。此外，在选择 Magnet Shuffle服务时，Spark Driver 可以选择优化以获得更好的负载均衡，而不是数据局部性。Spark Driver 可以查询可用 Magnet

Shuffle服务的负载，以便选择负载较少的服务。在我们的 Magnet
实现中，我们允许这种灵活的策略来选择 Magnet Shuffle服务的位置。因此，我们可以根据集群部署模式选择优化数据局部性、负载均衡或两者的组合。

3.5 处理掉队者和数据偏斜

任务掉队者和数据偏斜是实际数据处理中的常见问题，它们会减慢作业的执行速度。要在我们的环境中成为实用的解决方案，Magnet 需要能够处理这些潜在问题。

3.5.1 任务掉队者

当所有 map Task 在Shuffle map
阶段结束时完成，Shuffle块推送操作可能尚未完全完成。此时，有一批 map Task
刚开始推送块，也可能有掉队者无法足够快地推送块。与 Reduce Task
中的掉队者不同，我们在Shuffle map
阶段末尾经历的任何延迟都会直接影响作业的运行时间。如图 6
所示。为了缓解此类掉队者，Magnet 允许 Spark Driver
设定一个上限，表明它愿意等待块推送/合并操作的最长时间。在等待结束时，Spark Driver 通知所有选定的给定Shuffle服务的磁石Shuffle服务停止合并新的Shuffle块。这可能会导致在Shuffle Reduce 阶段开始时，一些块未能合并，然而它确保磁石可以提供推并Shuffle的大部分好处，同时将拖延者的负面影响限制在 Spark 应用程序的运行时间内。

3.5.2 数据偏斜

当一或多个Shuffle分区变得比其他分区明显大时，就会发生数据偏斜。在磁石中，如果我们尝试推送和合并这样的分区，我们可能最终会合并包含数十甚至数百吉字节数据的分区，这是不可取的。现有的 Spark 解决方案可以处理数据偏斜问题，例如自适应执行特性[14]。借助 Spark 自适应执行，在运行时收集有关每个Shuffle分区大小的数据，这些数据可用于检测数据偏斜。如果检

测到此类偏斜，触发Shuffle的操作符（如连接或分组）可能会将偏斜的分区划分为多个桶，以便将偏斜分区的计算分散到多个 Reduce Task
上。磁石与此类缓解偏斜的解决方案集成良好。当按照算法 1 划分 map Task 的Shuffle块为多个块时，磁石可以通过不将大于预定义阈值的Shuffle块包含在任何块中来修改算法。这样，磁石将合并所有正常分区，但跳过超出大小阈值的偏斜分区。通过 Spark
自适应执行，非偏斜的分区仍然会完整地获取，这可以从 Magnet
的推入合并Shuffle中受益。对于偏斜的分区，由于 Magnet
不会合并它们，且原始未合并的块仍然可用，因此它不会干扰 Spark
自适应执行的偏斜缓解解决方案。

我们在 Apache Spark 2.3 的基础上实现了 Magnet。由于修改特定于 Spark
内部实现而非面向用户的公共 API，现有的 Spark 应用程序可以在不改动代码的情况下从 Magnet 获益。为了达到最佳效率和可扩展性，我们还在实现中应用了一些优化。

4.1 并行化数据传输和任务执行

在并行化Shuffle数据获取和任务执行方面，Spark 优于传统的 Hadoop MapReduce 引擎。如图 7(a)所示，Hadoop MapReduce 通过一种称为“慢启动”的技术实现有限的并行化。它允许一些 reduce 任务开始获取Shuffle数据，而一些 map 任务仍在运行。这样，只有一部分 reduce 任务的Shuffle数据获取时间与 map 任务的执行时间重叠。在这方面，Spark 做得更好。Spark 不是让 map 任务和 reduce 任务重叠（在 DAG 执行引擎中不易管理），而是通过异步 RPC 实现并行化。使用单独的线程来获取远程Shuffle块和在获取的块上执行 reduce 任务。这两条线程像一对生产者和消费者一样，允许在Shuffle数据获取和减少任务执行之间更好地重叠。如图 7（b）所示。

在我们的 Magnet 实现中，我们采用类似的技术来实现并行数据传输和任务执行。在Shuffle map Task 端，通过利用专用线程池，我们将Shuffle块推送操作与 map Task 执行解耦。如图 7（c）所示。这允许在后续 mapper 执行时并行进行块推送操作。虽然Shuffle map Task 可能更加 CPU/内存密集，但块推送操作则更加磁盘/网络密集。这种并行化有助于更好地利用 Spark 执行器可用的计算资源。

在Shuffle Reduce Task 端，如果合并位于远程的 Magnet Shuffle服务上的Shuffle文件，获取将整个合并的 Shuffle 文件作为一个单独的数据块可能会导致不希望的获取延迟，因为我们在数据获取和减少任务执行之间无法实现太多并行化。为了解决这个问题，Magnet 在向其中追加数据块时将每个合并的 Shuffle 文件分割成多个 MB 大小的小片。合并的 Shuffle 文件内的切片边界作为单独的索引文件存储在磁盘上。Spark Driver 只需跟踪合并的 Shuffle 分区的粒度信息。合并的 Shuffle 文件如何进一步分割成小片仅由 Magnet Shuffle 服务通过索引文件跟踪。当 reduce 任务获取远程合并的 Shuffle 文件时，Magnet Shuffle 服务会回复小片的数量，以便客户端可以将对合并 Shuffle 文件的单次获取转换为对各个 MB 大小小片的多次获取。这种技术有助于在优化磁盘 I/O 与并行化数据传输和任务执行之间实现更好的平衡。

4.2 资源高效的实现

Magnet 的实现对于 Shuffle 服务端来说 CPU 和内存的开销极小。Magnet Shuffle 服务在块推送和合并操作中的唯一责任是接受远程推送的块并将其追加到相应的合并 Shuffle 文件中。这一低开销是通过以下方式实现的：

与[33]不同， Shuffle 服务端不执行排序。在Shuffle过程中，无论是 map 端排序还是 Reduce 端排序，都是在 Spark 执行器内部由 map Task 和 Reduce Task 执行的。

与[4, 41]不同，Magnet
Shuffle服务在合并操作期间不会在内存中缓存Shuffle块。唯一的块缓冲发生在 Spark
执行器内部，而 Magnet Shuffle服务直接将块合并到磁盘上。

[41]中的Shuffle优化是通过Shuffle服务从本地 mapper 拉取并合并Shuffle块来实现的。这需要在合并前将块缓存到内存中以提高磁盘效率。随着Shuffle服务中并发合并流的数量增加，内存需求也会增长。[41]中提到，对于 20 个并发合并流需要 6-8GB
的内存。在繁忙的生产集群中，如果并发度更高，这可能成为一个扩展瓶颈。Magnet 在 Spark
执行器中缓存块，这样就将内存需求分散到所有执行器上。Spark
执行器中有限的并发任务也使得内存占用保持在较低水平，使 Magnet 更具扩展性。[33, 4]中，合并操作利用Shuffle服务内每个分区的预写缓冲区来批量写入合并后的Shuffle数据。虽然这有助于减少合并期间的写操作数量，但内存需求带来了与[41]类似的扩展瓶颈。

4.3 优化磁盘 I/O

在第 3.2 节中，我们展示了 Magnet 如何批量读取Shuffle块以提高磁盘 I/O 效率。然而，这一改进要求在合并操作期间第二次编写大部分中间Shuffle数据。尽管这看起来代价高昂，但我们认为总体 I/O 效率还是有所提高的。不同于使用硬盘驱动器的小随机读取，小随机写入可以从多个缓存层次（如操作系统页面缓存和磁盘缓冲区）中受益。这些缓存将多个写入操作组合成一个，减少了放置在磁盘上的写入操作数量。因此，小随机写入能够实现比小随机读取更高的吞吐量。这也显示在我们第 5.2.3 节的基准测试中。通过批量读取Shuffle块和缓存批量处理写入操作，即使执行了双重写入，Magnet 也减少了Shuffle小Shuffle块的整体磁盘 I/O 操作。如第 3.5.2 节所述，对于大Shuffle块，我们跳过合并这些块。因此，它们不会产生双重写入的开销。实际上，由于 Magnet 的尽力而为特性，选择合适的操作系统 I/O 调度器来优先处理读取操作而不是写入操作，可以帮助减少双重写入的开销。我们还正在研究使用小型专用固态硬盘作为缓冲区，以便批量写入更多的块。与[33,
4]中的预写缓冲区方法相比，这提高了硬盘写入效率，且不会产生额外的内存开销。

在本节中，我们展示了 Magnet
的基准测试结果。通过结合合成型和生产型工作负载评估，我们展示了 Magnet
Shuffle服务能够有效提升磁盘 I/O 效率，缩短作业运行时间，同时保持资源高效利用。

5.1 评估设置

我们在两种不同的环境中对 Magnet
进行了评估。第一个是我们开发的一个分布式压力测试框架，用于对单个 Spark Shuffle服务的性能进行压力测试。它可以生成非常高的Shuffle块获取和推送操作的工作负载，以模拟繁忙的生产集群。更多细节见第 5.2.1 节。第二个是部署了 Magnet Shuffle服务的基准集群。如第 2.1
节所述，Spark Shuffle服务运行在底层的 YARN NodeManager 实例中。在这两种环境中，YARN NodeManager 实例都仅配置了 3GB 堆大小。此外，两种环境都使用具有 56 个 CPU 核心和每个 256GB RAM 的节点，通过 10Gbps 以太网链路连接。就存储而言，每个节点有 6
个硬盘驱动器，每个顺序读取的 I/O 速度为 100
MB/s。一些配备固态硬盘的节点也被用来评估不同存储介质的影响。

表 1：合成型Shuffle数据配置。它显示了要Shuffle的数据总量， map Task
的数量（相当于生成的Shuffle文件数）， Reduce Task
的数量（相当于每个Shuffle文件中的块数），以及单个块的大小。

5.2 合成工作负载评估

我们使用分布式压力测试框架来进行合成工作负载评估。通过模拟繁忙集群高峰时段接收的工作负载，对单个 Spark Shuffle服务进行压力测试，我们评估了与原生 Spark 相比，Magnet

Shuffle服务的完成时间、磁盘 I/O 吞吐量和资源占用情况。

5.2.1 压力测试框架

为了评估 Magnet Shuffle服务的特性，特别是在重Shuffle工作负载下的表现，我们开发了一个分布式压力测试框架，该框架能够模拟单个 Spark Shuffle服务会接收到的高负载。这样的压力测试框架使我们能够控制以下三个参数，以便即使在繁忙集群的高峰时段才能遇到的情况下，也能观察到 Spark Shuffle服务的性能。

 单个 Spark Shuffle服务会接收的并发连接数 单个Shuffle块的大小 中间Shuffle数据的总大小

此外，该框架能够对Shuffle块的获取和推送操作进行压力测试。前者将Shuffle块从单个 Spark Shuffle服务传输到多个客户端，而后者则将Shuffle块从多个客户端传输到单个Shuffle服务。通过这个框架，我们能够评估原生 Spark Shuffle服务以及 Magnet Shuffle服务。

在这个压力测试框架中，我们首先生成具有特定块大小和总大小的合成Shuffle数据。根据是评估块获取还是推送操作，这样的合成Shuffle数据可以在运行 Spark Shuffle服务的单个节点上生成，或者分布在运行客户端的多个节点上。生成合成Shuffle数据后，会启动一定数量的客户端，每个客户端都开始从/向 Spark
Shuffle服务获取或推送其部分的Shuffle块。这些客户端在如何获取或推送块方面执行与 Spark map 和规约任务相同的操作。每个客户端与 Spark
Shuffle服务建立多个连接。启动的客户端数量乘以每个客户端建立的连接数，成为 Spark Shuffle服务接收的并发连接数。

在我们的基准测试中，我们使用一个节点来运行 Spark Shuffle服务，20
个节点来运行客户端。我们生成了三组不同的合成Shuffle数据，如表 1
所示。这三组合成Shuffle数据总数据量相同，但块大小不同。

5.2.2 完成时间

我们想要评估的第一个指标是传输Shuffle数据的完成时间。在使用 Magnet
时，将中间Shuffle数据从 map Task 转移到 Reduce Task 涉及首先将Shuffle块推送到 Magnet Shuffle服务进行合并，然后从 Magnet Shuffle服务获取合并后的Shuffle块以供 Reduce Task

使用。在这里，我们评估了Shuffle块获取和推送操作的完成时间。

在这个基准测试中，我们对Shuffle块的获取和推送操作进行了三次运行，配置如表 1
所示。此外，每次运行使用了 200
个客户端，每个客户端有五个连接。因此，Shuffle服务接收到了 1000
个并发连接。作为比较，在生产集群的高峰时段，我们的 Spark
Shuffle服务接收到的并发连接数大致相同。

对于Shuffle块获取操作，随着块大小的增大，Shuffle服务的 I/O
操作逐渐从小随机读取转变为大量顺序读取。这带来了更好的 I/O 效率和更短的完成时间，如图
8(a)所示。至于Shuffle块推送操作，由于客户端从Shuffle文件中读取大块数据，不论块大小如何
，小块大小对Shuffle块推送操作的效率影响较小。这也如图
8(a)所示。从这个基准测试中，我们可以清楚地看到，Magnet 在从 HDD
读取时有效减轻了小Shuffle块的负面影响。通过单一Shuffle服务传输 150GB 的 10KB
块，与需要四小时的普通 Spark 相比，Magnet 只需略多于五分钟。

我们进一步比较了使用 HDD 和 SSD
完成Shuffle块获取操作所需的时间。与之前的基准测试类似，从单个Shuffle服务获取 150GB
的数据，数据块大小不同，使用了 1000 个并发连接。如图
8(b)所示，随着数据块大小的改变，SSD 表现出更为稳定的性能。这进一步证明了 Magnet
能够实现Shuffle数据传输的最佳磁盘效率，无论底层存储硬件如何。

5.2.3 磁盘 I/O

我们进一步评估了Shuffle块获取和推送操作的磁盘 I/O 效率。由于 HDD 受限于有限的 IOPS，我
们选择测量磁盘读写吞吐量，以比较Shuffle服务读取/写入块数据的速率。在执行Shuffle块获取
操作时，Shuffle服务主要从磁盘读取Shuffle块。在Shuffle块推送操作中，Shuffle服务主要将Shu
ffle块写入磁盘。

在此基准测试中，除了评估不同的数据块大小之外，我们还评估了Shuffle服务接收的并发连接数
量的影响。对于块获取操作，我们观察到Shuffle服务的磁盘读取速度随着块大小的增加，吞吐量
显著增加（图 9(a)）。这再次说明了用 HDD 进行小读取的低效性。此外，由于 Shuffle 块以随机
顺序只被读取一次，缓存不会有助于提高读取吞吐量。然而，随着并发连接数的增加，我们并没

有看到磁盘读取吞吐量的任何显著增加。所有三种块大小都是这种情况。理论上，连接数的增加意味着对可用网络带宽的更多利用。我们没有看到任何吞吐量增加的情况表明性能瓶颈在磁盘上，客户端无法充分利用可用的网络带宽。

图 8：使用 Magnet Shuffle 服务的 Shuffle 块推送操作与使用原生 Spark ESS 的 Shuffle 块获取操作的完成时间比较。(a)显示，块推送操作受小块大小影响不大。(b)显示，块获取操作需要如 SSD 这样的存储介质才能实现相同的效果。这验证了 Magnet 可以优化 Shuffle
数据传输，而不受底层存储硬件的影响。

对于块推送操作，随着块大小的改变，磁盘写入吞吐量更为稳定。处理块推送请求时，Magnet Shuffle 服务将单个块追加到合并的 Shuffle 文件中。与小随机读取不同，多级缓存（如操作系统页面缓存和磁盘缓冲）有助于提高性能。因此，小随机写入的写吞吐量受小块大小的影响要小得多。如图 9(b)所示。

图 9：随机读写操作对磁盘吞吐量的比较。(a)显示，小随机读写块大小会严重影响硬盘读取吞吐量。(b)显示，硬盘写入吞吐量在进行块推送操作时更为稳定。在这两种情况下，连接数的增加并未带来吞吐量的提升，表明性能瓶颈在于磁盘而非网络。

5.2.4 资源使用情况

我们还评估了执行块推送操作时 Magnet
随机读写服务的资源使用情况。我们生产集群中部署的常规 Spark
随机读写服务能够处理每天数百 PB
的随机读写数据。这在很大程度上归功于随机读写服务在执行随机读写块获取操作时对 CPU
和内存的低资源占用。为了达到同样的效果，Magnet 随机读写服务也需要有较低的资源占用。

在这次基准测试中，我们像之前一样，评估了三种不同块大小的随机读写获取和推送操作。对于每次运行，随机读写服务接收 1000
个并发连接，以模拟随机读写服务在高峰时段的场景。在数据传输过程中，Magnet
随机读写服务的 CPU 消耗与常规 Spark 随机读写服务相似。在三次不同块大小的运行中，其平均 CPU 虚拟核心占用率为 20-50%。这是合理的，因为 shuffle
块推送操作是磁盘密集型操作，类似于块获取操作。就内存消耗而言，Magnet
Shuffle服务的占用空间与原生 Spark Shuffle服务相似，堆上和堆外内存的平均消耗约为
300MB。此外，其内存使用不受块大小的影响。这在很大程度上是由于第 4.2
节中引入的实现优化所致。

5.3 生产工作负载评估

我们进一步用几个选定的实际工作负载对 Magnet 进行评估。该评估是在一个包含 200 个节点的基准集群上进行的。我们启动了 200 个 Spark 执行器，每个执行器拥有 5GB 内存和 2 个虚拟核心。我们在评估中使用了 3 个生产作业，分别代表小型作业（工作负载 1）、中等 CPU 密集型作业（工作负载 2）、大型Shuffle密集作业（工作负载 3）。这些工作负载具有以下特点：

 工作负载 1 是一个简短的 SQL 查询，其数据Shuffle量少于 100GB 工作负载 2 的数据Shuffle量约为 400GB，并且在Shuffle map 和 Reduce Task 中都占用大量 CPU 资源 工作负载 3 则更加侧重于 I/O，其数据Shuffle量约为 800GB

我们测量了使用和不使用 Magnet Shuffle服务时这些工作负载的三个指标：1）所有Shuffle map 阶段的端到端运行时间，2）所有 Reduce 阶段的端到端运行时间，以及 3）总任务运行时间。结果显示在表 2 中。对于工作负载 1，由于Shuffle开销较低，它从 Magnet 中获益不多。运行时间的比较进一步验证了 Magnet 没有引入太多开销，从而不会导致性能下降。对于工作负载 2，总作业运行时间减少了 45%，而使用 Magnet 时， Reduce 阶段的运行时间和总任务运行时间减少了 68%。这验证了即使对于 CPU 密集型作业，通过优化Shuffle操作，Magnet 也能提供显著改进。对于工作负载 3，即 I/O 密集型的Shuffle作业，Magnet 显著减少了作业运行时间。具体来说，Magnet 将Shuffle Reduce 阶段的运行时间减少了 81%，同时保持Shuffle map 阶段的运行时间基本不受影响。

除了对单个工作负载的 Magnet 性能进行测量外，我们还同时对工作负载 2 和工作负载 3 进行了基准测试，每个使用 100 个 Spark 执行器。这是为了测量 Magnet 在混合 CPU 密集型和 I/O 密集型作业下的性能，以及当 Magnet Shuffle服务同时处理块推送和获取请求时的表现。从表 2 中我们可以看出，结合多个工作负载时，Magnet 的性能提升更为显著。这表明，尽管标准的 Spark Shuffle服务在增加Shuffle工作负载下性能会下降，但 Magnet 仍能持续实现卓越的Shuffle性能。

最近，针对分布式数据处理框架的Shuffle优化研究引起了极大的兴趣。与 Magnet 不同，在

Magnet 中， mapper
将Shuffle块推送到远程Shuffle服务以进行合并，Riffle[41]则通过Shuffle服务从本地 mapper 拉取Shuffle块来执行块合并操作。与 Magnet 相比，它有三个缺点：1）基于拉取的合并服务在合并前需要在内存中缓存数据块以提高磁盘效率。随着 shuffle 服务中并发合并流的数量增加，内存需求也会增长。这可能在繁忙的生产集群中成为扩展瓶颈。而 Magnet 在 Spark 执行器中缓存数据块，这样就将内存需求分散到所有执行器上。Spark 执行器中有限的并发任务也使得内存占用保持在很低的水平，使 Magnet 更具可扩展性。2）本地合并服务不会重新定位 shuffle 数据块，因此它不能为 reducer 提供更好的 shuffle 数据局部性。3）同样，本地合并服务不会在不同节点上复制 shuffle 数据块，因此不能帮助提高容错能力。

Sailfish[33]/Cosco[4]是另外两种合并 shuffle 数据块的解决方案。Sailfish 建立在 I 文件抽象的基础上，这是对 KFS[8]的扩展，它将 Hadoop MapReduce 的每个分区的 shuffle 数据聚合成 I 文件，每个 I 文件包含多个块。Cosco[4]是为 Spark 实现 Sailfish 的程序。这两种解决方案都旨在实现非聚合的集群部署。计算引擎将 shuffle 中间数据的管理委托给外部存储系统，Sailfish 使用 KFS，而 Cosco 则使用 Facebook 的非聚合存储集群。这包括存储 shuffle 数据、提供容错以及跟踪 shuffle 数据块的位置元数据。另一方面，Magnet 与 Spark 原生 shuffle 集成，此时 Spark 仍然管理 shuffle 的所有方面。旗鱼/中远海运的委托Shuffle方法与 Magnet 相比有两个缺点：1）对外部存储系统的依赖使其部署更具限制性。在已存在专用存储层的云环境中，可能无法大规模部署自定义存储解决方案。随着 LinkedIn 从本地集群迁移到微软 Azure，设计一种可在两种环境中灵活部署的解决方案非常重要。2）使用委托Shuffle时，Spark 无法利用隐藏在外部系统中的Shuffle元数据来智能地调度 reducer 或处理Shuffle获取失败。Magnet 与 Spark 原生Shuffle集成，通过提供更好的Shuffle数据局部性来提高性能。虽然集成的Shuffle方法提供了更灵活的部署策略和原生Shuffle性能，但委托Shuffle方法与特定引擎的耦合度较低，因此可以更容易地扩展以支持多个计算引擎。

iShuffle 是另一项通过写时Shuffle技术优化 Hadoop MapReduce Shuffle的工作。它将 mapper 生成的Shuffle块推送到 reducer。与 Magnet 相比，iShuffle 受限于慢速节点的影响，并且不能帮助提高Shuffle的可靠性。此外，由于 Hadoop MapReduce 采用一级单体调度机制，而不是 YARN 上的两级调度机制的 Spark，iShuffle 也无法有效地调度 reduce 任务以利用多租户集群中的数据局部性。

还有一些其他研究旨在通过不同的方法改进Shuffle（shuffle）。HD shuffle[31] 提出了一种新的Shuffle算法，它将单个大型扇入扇出Shuffle划分为多个有界扇入扇出的Shuffle。虽然这有助于提高磁盘效率，但它引入了额外的Shuffle操作，可能并不理想。MapReduce Online[20] 提出了一种基于推送的机制来优化在线聚合和连续查询。它适用于内存中的Shuffle，并且不是为大作业设计的。Hadoop-A[39] 提出了一种利用 RDMA 优化Shuffle的方法，而 MemVerge 的 Splash[42] 是一个利用 PMEM
的解决方案。这些解决方案的优势与非商品硬件相结合。值得注意的是，Riffle[41]、Sailfish[33] 和 Cosco[4] 是迄今为止唯一已知在生产环境中大规模部署的解决方案。

在本文中，我们介绍了 Magnet，一种利用推送合并Shuffle来提高 Spark
中Shuffle操作效率、可靠性和可扩展性的 Spark Shuffle服务。Magnet
通过合并Shuffle块，在Shuffle写入和读取路径上实现了优化的磁盘 I/O。它还通过以原始形式和优化形式复制Shuffle数据，进一步减少了与Shuffle相关的故障。Magnet 还改善了Shuffle
Reduce Task 的数据局部性，这进一步提高了 Spark
中Shuffle的效率和可靠性。根据我们的评估，我们展示了 Magnet 有助于缓解 Spark
Shuffle操作的几个现有问题，并缩短了作业完成时间。

我们感谢匿名 VLDB 审稿人提供的宝贵且建设性的反馈。我们感谢 Eric Baldeschwieler、Brian Cho、Walaa Moustafa、Yuval Degani、Khai Tran、Sriram Rao、Tyson Condie 和 Phil Bernstein 对初稿的广泛评论以及关于此话题的深入讨论。我们还要感谢 LinkedIn BDP 管理团队，特别是 Eric Baldeschwieler、Vasanth Raja-mani、Zhe Zhang 和 Sunitha Beeram 的支持。

[1] Apache Hadoop. http://hadoop.apache.org (Retrieved 02/20/2020).

[2] Apache spark the fastest open source engine for sorting a petabyte.
https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html (Retrieved 02/20/2020).

[3] Building the next version of our infrastructure. https://engineering.linkedin.com/blog/2019/building-next-infra (Retrieved 05/15/2020).

[4] Cosco: An efficient facebook-scale shuffle service. https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service (Retrieved 02/20/2020).

[5] Create shuffle service for external block storage. https://issues.apache.org/jira/browse/SPARK-3796 (Retrieved 02/20/2020).

[6] Dr. elephant for monitoring and tuning apache spark jobs on hadoop. https://databricks.com/session/dr-elephant-for-monitoring-and-tuning-apache-spark-jobs-on-hadoop (Retrieved 02/20/2020).

[7] Introduce pluggable shuffle service architecture. https://issues.apache.org/jira/browse/FLINK-10653 (Retrieved 02/20/2020).

[8] KFS. https://code.google.com/archive/p/kosmosfs/ (Retrieved 02/20/2020).

[9] Making apache spark effortless for all of uber. https://eng.uber.com/uscs-apache-spark/ (Retrieved 02/20/2020).

[10] Netflix: Integrating spark at petabyte scale. https://conferences.oreilly.com/strata/big-data- conference-ny-2015/public/schedule/detail/43373 (Retrieved 02/20/2020).

[11] plugin for generic shuffle service. https://issues.apache.org/jira/browse/MAPREDUCE-4049 (Retrieved 02/20/2020).

[12] Spark data locality documentation. https://spark.apache.org/docs/latest/tuning.html#data-locality (Retrieved 02/20/2020).

[13] Spark dynamic resource allocation documentation. https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation

(Retrieved 02/20/2020).

[14] Spark sql adaptive execution at 100 tb. https://software.intel.com/en-us/articles/spark-sql-adaptive-execution-at-100-tb (Retrieved 02/20/2020).

[15] Taking advantage of a disaggregated storage and compute architecture. https://databricks.com/session/taking-advantage-of-a-disaggregated-storage-and-compute-architecture (Retrieved 02/20/2020).

[16] Tuning apache spark for large-scale workloads. https://databricks.com/session/tuning-apache-spark-for-large-scale-workloads (Retrieved 02/20/2020).

[17] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. USENIX HotOS, 2011.

[18] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015.

[19] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. Bulletin IEEE Comput. Soc. Tech. Committee Data Eng, 38(4):28–38, 2015.

[20] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. NSDI 2010, 10(4), 2010.

[21] A. Davidson and A. Or. Optimizing shuffle performance in spark. University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep, 2013.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Commun. ACM, 5(1):107–113, 2008.

[23] R. C. Gon¸calves, J. Pereira, and R. Jim´enez-Peris. An rdma middleware for asynchronous

multi-stage shuffling in analytical processing. IFIP International Conference on Distributed Applications and Interoperable Systems, pages 61–74, 2016.

[24] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of nand flash memory. FAST, 2012.

[25] Y. Guo, J. Rao, D. Cheng, and X. Zhou. ishuffle: Improving hadoop performance with shuffle-on-write. IEEE Transactions on Parallel and Distributed Systems, 28(6):1649–1662, 2016.

[26] Z. Guo, G. Fox, and M. Zhou. Investigation of data locality in mapreduce. 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 419–426, 2012.

[27] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. Cidr, pages 261–272, 2011.

[28] V. Kasavajhala. Solid state drive vs. hard disk drive price and performance study. Proc. Dell Tech. White Paper, pages 8–9, 2011.

[29] B. S. Kim, J. Choi, and S. L. Min. Design tradeoffs for SSD reliability. 17th USENIX Conference on File and Storage Technologies (FAST 19), pages 281–294, 2019.

[30] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. Proceedings of the 24th ACM Symposium on Operating Systems Principles, pages 69–84, 2013.

[31] S. Qiao, A. Nicoara, J. Sun, M. Friedman, H. Patel, and J. Ekanayake. Hyper dimension shuffle: Efficient data repartition at petabyte scale in scope. PVLDB, 12(10):1113–1125, 2019.

[32] N. Rana and S. Deshmukh. Shuffle performance in apache spark. International Journal of Engineering Research and Technology, pages 177–180, 2015.

[33] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. Proceedings of the 3rd ACM Symposium on Cloud Computing, pages 1–14, 2012.

[34] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. Proceedings of the 8th ACM European Conference on Computer Systems, pages 351–364, 2013.

[35] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: Sql on everything. Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1802–1813, 2019.

[36] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. IEEE Data Eng. Bull., 40(1):38–49, 2017.

[37] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. Proceedings of the 4th annual Symposium on Cloud Computing, pages 1–16, 2013.

[38] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. Optimizing load balancing and data-locality with data-aware scheduling. 2014 IEEE International Conference on Big Data, pages 119–128, 2014.

[39] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–10, 2011.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012.

[41] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman. Riffle: optimized shuffle service for large-scale data analytics. Proceedings of the 13th EuroSys Conference, pages 1–15, 2018.

[42] P. Zhuang, K. Huang, Y. Zhao, W. Kang, H. Wang, Y. Li, and J. Yu. Shuffle manager in a distributed memory object architecture, 2020. US Patent App. 16/372,161.