Magnet: Push-based Shuffle Service for Large-scale Data Processing

Min Shen
LinkedIn
mshen@linkedin.com

Ye Zhou
LinkedIn
yezhou@linkedin.com

Chandni Singh
LinkedIn
chsingh@linkedin.com

ABSTRACT

Over the past decade, Apache Spark has become the most prevalent compute engine for large scale data processing. Similar to other compute engines based on the MapReduce compute paradigm, the shuffle operation, namely the all-to-all transfer of the intermediate data, plays an important role in Spark. At LinkedIn, with the rapid growth of the data size and the scale of the Spark deployment, the shuffle operation is quickly becoming a bottleneck of further scaling the infrastructure. This has led to overall job slowness and late failures for long running jobs. This not only impacts developer productivity for addressing such slowness and failures, but also results in measurable high operational cost of infrastructure.

In this work, we present the main bottlenecks impacting shuffle scalability. We further propose Magnet, a novel shuffle mechanism that can scale to handle petabytes of daily shuffled data and clusters with thousands of nodes. Magnet is designed to work with both on-prem and cloud-based cluster deployments. It addresses a key shuffle scalability bottleneck by merging fragmented intermediate shuffle data into large blocks. Magnet provides further improvements by co-locating merged blocks with the reduce tasks. Our benchmarks show that Magnet drastically improves the shuffle performance independent of the underlying hardware. With Magnet, we have seen close to 30% reduction in the end-to-end runtime of LinkedIn's production Spark jobs. Furthermore, Magnet improves on user productivity as well by removing the shuffle related tuning burden from users.

PVLDB Reference Format:
M. Shen, Y. Zhou and C. Singh. Magnet: Push-based Shuffle Service for Large-scale Data Processing. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Distributed data processing frameworks such as Hadoop [1] and Spark [40] have gained major popularity over the past decade for large-scale data analysis use cases. Based on the MapReduce computing paradigm [22] and leveraging a large suite of commodity machines, these distributed data processing frameworks have shown good characteristics of scalability and broad applicability to a diverse collection of use cases, ranging from data analytics to machine learning and AI. In more recent years, a collection of modern computation engines, such as Spark SQL [18], Presto [35], and Flink [19], have emerged and gone mainstream. Different from Hadoop MapReduce, these modern computation engines leverage SQL optimizers to drastically optimize the computation logic specified by the users, before handing over to DAG execution engines to execute the optimized operations. Take Spark as an example (Figure 1). Suppose the user wants to perform an inner join between the *job_post_view* table and the *job_dimension* table before filtering the joined results based on certain conditions. In this example, the former table contains tracking information for which member viewed which job post on LinkedIn platform, and the latter contains the detailed information of each job post. Spark would recognize that it can optimize this query by pushing down the filter condition before the join operation (Figure 1(a)). Spark's DAG execution engine would then take this optimized compute plan and convert it into one or more *jobs*. Each job consists of a DAG of *stages*, representing the lineage of how the data gets transformed to produce the final results of current job (Figure 1(b)). The intermediate data between stages is transferred via the shuffle operation.

The shuffle operation, where intermediate data gets transferred via all-to-all connections between the map and reduce tasks of the corresponding stages, is key to the MapReduce computing paradigm [22]. Although the basic concept of the shuffle operation is straightforward, different frameworks have taken different approaches to implement it. Some frameworks such as Presto [35] and Flink streaming [19] would materialize intermediate shuffle data in memory for low latency needs, while others such as Spark [40] and Flink batch [19] would materialize it on local disks for better fault-tolerance. When materializing the intermediate shuffle data on disks, there are hash-based solutions, where each map task produces separate files for each reduce task, or sort-based solutions, where the map task's output is sorted by the hashed value of the partition keys and materialized as a single file. While the sort-based shuffle might incur the overhead of sorting, it is a more performant and scalable solution when the size of the intermediate shuffle data is large [21, 32]. Furthermore, frameworks such as Spark and Flink

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

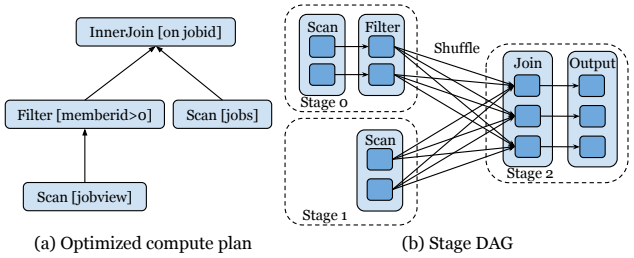


Figure 1: Compute logical plan and associated stage DAG of a Spark query.

start to adopt external shuffle services [5, 7, 11] to serve materialized intermediate shuffle data, in order to achieve better fault-tolerance and performance isolation. With the recent improvements in networking and storage hardware, we have also seen solutions [36, 15] that would materialize the intermediate shuffle data in disaggregated storage instead of local storage. In addition, we have also seen solutions [20, 23] that would bypass materializing the intermediate shuffle data, where the map tasks' output is directly pushed to the reduce tasks to achieve low latencies. Such a diversity in the shuffle implementation provides a rich optimization space for the shuffle operation in these computation engines. In fact, improving shuffle in Spark was the key for it to win the Sort Benchmark in the past [2].

At LinkedIn, as a global professional social network company, a very large amount of batch analysis and machine learning jobs run on our production Spark clusters spanning thousands of nodes on a daily basis. This leads to petabytes of data being shuffled by Spark each day. When processing such a massive amount of data on clusters of our scale, the shuffle operation becomes very critical to the operations of the Spark infrastructure. Spark materializes the shuffle data on disks in a sort-based fashion and serves it with external shuffle services. While this provides a good balance of fault-tolerance and performance, the ongoing fast growth of Spark workloads at LinkedIn still poses multiple challenges.

First, the requirement of establishing all-to-all connections to transfer the shuffle data between the map and reduce tasks is posing reliability issues. In clusters with thousands of nodes, intermittent issues with individual node availability can be common. During the peak hours, the increased shuffle workloads could also stress the deployed shuffle services, further increasing the likelihood of connection failures.

Second, the disk I/O operations generated during shuffle present efficiency issues. The materialized Spark shuffle data is portioned into shuffle blocks, which are fetched individually in a random order. The size of these blocks are usually small. In fact, the average block size in LinkedIn's Spark clusters is only around 10s of KBs. Billions of such blocks are read on our clusters on a daily basis, which can severely stress the disks if served from HDDs. The small random disk reads, combined with other associated overheads such as small network I/Os, lead to increased latency in fetching shuffle data. Around 15% of the total Spark computation resources on our clusters are wasted due to this latency.

Finally, as pointed out in [41], the shrink of the average block size as the shuffle data size grows also introduces a scalability issue. As the Spark workload generally trends towards processing more data, we have observed the above mentioned efficiency issue gradually getting worse in our clusters over the past year. Some abusive Spark applications due to misconfiguration further exacerbate this problem.

While solutions such as storing shuffle data on SSDs can help to alleviate these problems, switching out HDDs with SSDs at LinkedIn scale might not be practical. More details on this are discussed in Section 2.2. In addition, for cloud-based cluster deployments that leverages disaggregated storage, it would also suffer from small reads due to network overhead. To tackle these challenges, we propose an alternative shuffle mechanism named Magnet. With Magnet, we would push the fragmented shuffle blocks produced by every map task to remote shuffle services, and merge them into large chunks per shuffle partition in an opportunistic fashion. Some of its benefits are highlighted below:

- Magnet opportunistically merges fragmented intermediate shuffle data into large blocks and co-locates them with the reduce tasks. This allows Magnet to significantly improve the efficiency of the shuffle operation and decrease the job end-to-end runtime, independent of the underlying storage hardware.
- Magnet adopts a hybrid approach where both merged and unmerged shuffle data can serve as input to reduce tasks. This helps to improve reliability during shuffle.
- Magnet is designed to work well in both on-prem or cloud-based deployments, and can scale to handle petabytes of daily shuffled data and clusters with thousands of nodes.

The rest of this paper is organized as follows: Section 2 introduces the background of the Spark shuffle operation and discusses the existing issues. Section 3 presents the detailed designs of Magnet. Section 4 shows some optimizations adopted in the implementation of Magnet. Section 5 gives the evaluation setups, key results, and analysis. Section 6 talks about related works. We conclude this paper in Section 7.

2. BACKGROUND AND MOTIVATION

In this section, we motivate and provide the background for Magnet. Section 2.1 reviews the current Spark shuffle operation. Sections 2.2-2.4 discuss the major shuffle issues we have encountered with operating Spark infrastructure at very-large scale.

2.1 Current Spark Shuffle Operation

As mentioned earlier, the intermediate data between stages is transferred via the shuffle operation. In Spark, the way shuffle is performed varies slightly based on the deployment mode. At LinkedIn, we deploy Spark on YARN [37] and leverage the external shuffle service [5] to manage the shuffle data. Such a deployment mode is also widely adopted across the industry, with adoptions from companies having some of the largest Spark deployments, including Netflix [10], Uber [9], and Facebook [16]. With such a Spark deployment, the shuffle operation in Spark works as illustrated in Figure 2:

1. Each Spark executor upon starting up would register with the Spark External Shuffle Service (ESS) that is located on the same node. Such registrations would allow the Spark ESS to know about the locations of the materialized shuffle data produced by local map tasks

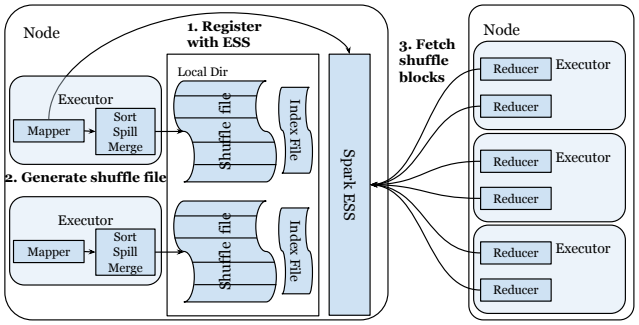


Figure 2: Illustration of the 3 main steps in Spark shuffle operation.

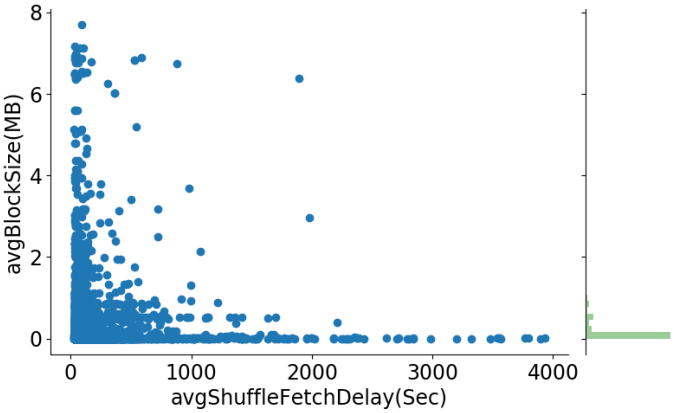


Figure 3: Scatter plot of sampled 5000 shuffle reduce stages with significant delays. The graph shows the per task average shuffle fetch delay and the average shuffle block size. The bar chart to the right of the scatter plot shows the distribution of the shuffle block size within this dataset. Majority of the data points have a small block size.

from each registered executor. Note that the Spark ESS instances are external to the Spark executors and shared across potentially many Spark applications.

2. Each task within a shuffle map stage would process its portion of the data. At the end of the map task, it would produce a pair of files, one for the shuffle data and another to index shuffle blocks in the former. To do so, the map task would sort all the transformed records according to the hashed value of the partition keys. During this process, the map task might spill the intermediate data to disk if it cannot sort the entire data in memory. Once sorted, the shuffle data file is generated, where all records belonging to the same shuffle partition are grouped together into a shuffle block. The matching shuffle index file is also generated which records the block boundary offset.

3. When reduce tasks of the next stage start to run, they will query the Spark driver for locations of their input shuffle blocks. Once this information becomes available, each reduce task will establish connections to corresponding Spark ESS instances in order to fetch their input data. The Spark ESS, upon receiving such a request, would skip to the corresponding block data in the shuffle data file leveraging the shuffle index file, read it from disk, and send it back to the reduce task.

By materializing the shuffle data to disks in a sort-based fashion, Spark shuffle operation achieves a reasonable balance between performance and fault-tolerance. Furthermore, by decoupling from Spark executors, Spark ESS brings additional benefits to the Spark shuffle operation: 1) The Spark ESS can serve the shuffle block even when Spark executors are experiencing GC pause. 2) The shuffle blocks can still be served even if the Spark executor generating them is gone. 3) The idle Spark executors can be freed up to save cluster compute resources. However, in our operation of Spark infrastructure at LinkedIn scale, we still observed multiple issues with shuffle that are becoming a potential bottleneck for the infrastructure's reliability, efficiency, and scalability. We will discuss these issues in the following sub-sections.

2.2 Inefficient Disk I/O during Shuffle

One of the biggest issues we have observed with Spark shuffle operation is the inefficient disk I/O due to small shuffle blocks. Since the Spark ESS would only read one shuffle block for each fetch request, the average size of the shuffle block would dictate the average amount of data read per disk read. At LinkedIn, we mostly use HDDs for intermediate shuffle data storage in our Spark clusters. This is due to the cost effectiveness of HDDs compared with SSDs for large-scale storage needs [28, 24], as well as SSD's prone to wear-out when used for temporary data storage such as shuffle data [29]. For HDD, serving a large number of small random reads would be subject to its IOPS (I/O operations per second) limitations. Furthermore, the shuffle data is usually read only once and the individual shuffle blocks are accessed in a random order. Such a data access pattern means that no types of caching would help to improve the disk I/O efficiency. The combination of HDD's limited IOPS and the shuffle data's access pattern lead to a low disk throughput and extended delays for fetching shuffle blocks.

In LinkedIn's production Spark clusters, although the average daily shuffled data size is big, reaching a few petabytes per day by the end of 2019, the average daily shuffled block count is also very high (10s of billions). This leads to a small average shuffle block size of only 10s of KBs. The small block size is leading to increasingly delayed shuffle data fetch. This is validated by the data we gathered. Figure 3 plots the correlation between the average shuffle block size and the per task average shuffle fetch delay for 5000 shuffle reduce stages with significant delays ($>$ 30 seconds per task). These stages are sampled from the production Spark jobs run in our clusters in April 2019. The graph further shows the distribution of the block size of these 5000 stages. From this graph, we can observe that majority of the shuffle reduce stages with significant shuffle fetch delays are ones with small block size.

Although the small shuffle block size seems to be addressable via properly configuring the number of map and/or reduce tasks, we argue that it is not easily achievable. There do exist misconfigured Spark applications in our clusters, where the number of reducers is set too large for the amount of data being shuffled. In such cases, parameter auto-configuration solutions such as [27, 6, 14] can help to fix the

problem, by either reducing the number of reducers or increasing the split size to reduce the number of mappers. However, there exist "tuning dilemmas" where changing the parameters to improve shuffle performance might negatively impact other aspects of the job. This is mostly due to increased amount of data processed per map/reduce task as a result of parameter changes. In addition, such "tuning dilemmas" would lead to increased small shuffle blocks.

As pointed out in [41], for a shuffle with M mappers and R reducers, if the amount of data processed per task stays the same, the number of shuffle blocks ($M * R$) would grow quadratically as the size of the shuffle data D grows. This would lead to the reduction of the shuffle block size ($\frac{D}{M*R}$) as D grows. Based on our experiences supporting thousands of Spark users at LinkedIn, keeping the amount of data processed per task relatively constant is a proven practice when the workload increases. By scaling a Spark application horizontally (more executors) instead of vertically (larger executors), it avoids the increased difficulty in acquiring executor containers in a multi-tenancy cluster. It also prevents the decreased resource utilization due to resource over-provisioning. At LinkedIn, due to the growing number of members joining the platform and the need of building more complex AI models, the analytics and machine learning workloads are processing more data than before. Based on the above analysis, such a need of processing more data would lead to the inevitable reduction of shuffle block size.

2.3 Reliability of Shuffle all-to-all Connections

Another common issue we observed is with the reliability of establishing the RPC connections with Spark ESS during shuffle. As introduced earlier, in the shuffle reduce stage, each task needs to establish connections to all of the remote shuffle services hosting its task input. For a shuffle of M mappers and R reducers, in theory a total of $M * R$ connections need to be established. In practice, reducers on the same executor share one outgoing connection per destination, and mappers registered with the same Spark ESS share one incoming connection per source. So, for a Spark application using S Spark shuffle services and E executors, up to $S * E$ connections would still be needed. In a large-scale deployment of Spark infrastructure, both S and E can be up to 1000. For a large-scale Spark cluster, intermittent node availability issues can happen. In addition, because the Spark ESS is a shared service, when a Spark ESS gets impacted by an abusive job or receives increased shuffle workloads during peak hours, it could also be stressed and have a reduced availability. When the reduce tasks experience failures in establishing connections to remote Spark ESS, it would immediately fail the entire shuffle reduce stage, leading to retries of the previous stages to regenerate the shuffle data that cannot be fetched earlier. Such retries can be very expensive, and have caused delays in Spark application runtime leading to production flow SLA breakage.

2.4 Placement of Reduce Tasks

A study [17] claims that data locality is no longer important in data center computing, and others [30, 26, 38] are still showing the benefits data locality can provide. While it is true that the speed of network has increased significantly in the past decade, we observe that due to limitations of spindle disks' IOPS as discussed in Section 2.2, we often cannot saturate the network bandwidth. This is validated in a benchmark we performed in Section 5.2. In addition, if the shuffle block is available locally to the reduce task, the task can directly read it from disk, bypassing the shuffle service. This also helps to reduce the number of RPC connections during shuffle. Although shuffle data locality can provide such benefits, the current shuffle mechanism in Spark would lead to little data locality for the reduce tasks, as their task input data is scattered across all the map tasks.

3. SYSTEM DESIGN

With these issues described in Section 2, we present Magnet, an alternative shuffle mechanism for Spark. Magnet aims at keeping the current fault-tolerance benefits of the current Spark shuffle operation, which materializes intermediate shuffle data in a sort-based fashion, while overcoming the above mentioned issues. In designing Magnet, we have to overcome several challenges.

First, Magnet needs to improve the disk I/O efficiency during the shuffle operation. It should avoid reading the individual small shuffle blocks from disks which hurts disk throughput. Second, Magnet should help to alleviate the potential Spark ESS connection failure issue. This is important for improving the overall reliability of the shuffle operation in large-scale Spark clusters. Third, Magnet needs to be able to cope with potential stragglers and data skews, which can be common in large-scale clusters with real-world production workload. Finally, Magnet needs to achieve these benefits without incurring much memory or CPU overhead. This is essential in making Magnet a scalable solution to handle clusters with thousands of nodes and petabytes of daily shuffled data.

3.1 Solution Overview

Figure 4 shows the architecture of Magnet. Four existing components in Spark (Spark driver, Spark ESS, Spark shuffle map task, and Spark shuffle reduce task) are extended with additional behaviors in our design. More details are covered in later subsections.

- Spark driver would coordinate the entire shuffle operation between the map tasks, reduce tasks, and the shuffle services. (Step 1, 6, 7 in Figure 4)
- The shuffle map tasks now handles the additional preparation of their materialized shuffle data for pushing to remote shuffle services. (Step 2, 3 in Figure 4)
- The Magnet shuffle service, which is an enhanced Spark ESS, would accept remotely pushed shuffle blocks and merge them into the corresponding merged shuffle file for each unique shuffle partition. (Step 4 in Figure 4)
- The shuffle reduce tasks now take advantage of these merged shuffle files, which are often co-located with the tasks themselves, in order to improve the efficiency of fetching their task inputs. (Step 8 in Figure 4)

Some key features of Magnet are briefly described below.

Push-Merge Shuffle - Magnet adopts a push-merge shuffle mechanism, where the mapper generated shuffle data gets pushed to remote Magnet shuffle services to be merged per shuffle partition. This allows Magnet to convert random reads of small shuffle blocks into sequential reads of MB-sized chunks. In addition, this push operation is decoupled

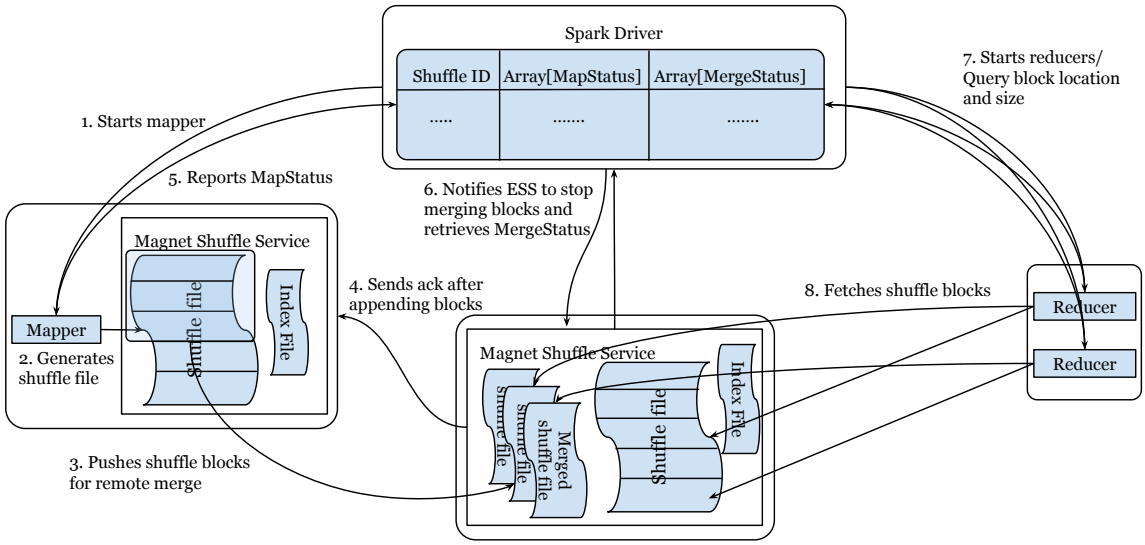


Figure 4: Illustration of Magnet architecture, with numbered steps explained in Section 3.1.

from the mappers, so that it does not add to the map task's runtime or lead to map task failures if the operation fails.

Best-effort Approach - Magnet does not require a perfect completion of the block push operation. By performing push-merge shuffle, Magnet effectively replicates the shuffle data. Magnet allows reducers to fetch both merged and unmerged shuffle data as task input. This allows Magnet to tolerate partial completion of the block push operation.

Flexible Deployment Strategy - Magnet integrates with Spark native shuffle by building on top of it. This enables Magnet to be deployed in both on-prem clusters with co-located compute and storage nodes and cloud-based clusters with disaggregated storage layer. In the former case, with the majority of each reduce task's input merged at one location, Magnet exploits such locality preference to schedule reduce tasks and achieves better reducer data locality. In the latter case, instead of data locality, Magnet can optimize for better load balancing by selecting remote shuffle services that are less loaded.

Stragglers/Data Skews Mitigation - Magnet can handle stragglers and data skews. Since Magnet can tolerate partial completion of block push operations, it can mitigate stragglers and data skews by either halting slow push operations, or skipping pushing large/skewed blocks.

3.2 Push-Merge Shuffle

In order to improve disk I/O efficiency, we need to either increase the amount of data read per I/O operation or switch to using storage mediums that are more optimized for small random reads such as SSDs or PMEM. However, as pointed out in Section 2.2, application parameter tuning cannot effectively increase the shuffle block size due to "tuning dilemmas", and SSDs or PMEM are not practical enough to store the intermediate shuffle data at scale. [33, 41] proposed solutions that would merge shuffle blocks belonging to the same shuffle partition in order to create a larger chunk. Such techniques are shown to effectively improve the disk I/O efficiency during shuffle. Magnet adopts a push-merge shuffle mechanism, which also leverages the shuffle block merge technique. Compared with prior works in [33, 41], Magnet achieves better fault-tolerance, reducer data locality, and resource-efficiency. More details on this comparison are given in Section 6.

3.2.1 Preparing Blocks for Remote Push

One principle we hold in designing the block push operation in Magnet is to incur as little CPU/memory overhead on the shuffle service as possible. As mentioned earlier, the Spark ESS is a shared service across all Spark applications in the cluster. For a Spark on YARN deployment, the resources allocated to the Spark ESS are usually orders of magnitude less than the resources available for Spark executor containers. Incurring heavy CPU/memory overhead on the shuffle service would impact its scalability. In [41], the Spark ESS needs to buffer multiple MB-sized data chunks from local shuffle files in memory in order to efficiently merge blocks. In [33], the mapper produced records are not materialized locally, but sent to per shuffle partition write ahead buffers in remote shuffle services. These buffered records are potentially sorted on the shuffle service before materialized into external distributed file systems. Both of these approaches are not ideal for ensuring low CPU/memory overhead in shuffle services.

In Magnet, we keep the current shuffle materialization behavior, which materializes the shuffle data in a sort-based fashion. Once a map task produces the shuffle files, it would divide the blocks in the shuffle data file into MB-sized chunks, each to be pushed to a remote Magnet shuffle service to be merged. With Magnet, the Spark driver determines a list of Magnet shuffle services the map tasks of a given shuffle should work with. Leveraging this information, each map task can consistently decide the mapping from shuffle blocks to chunks, and from chunks to Magnet shuffle services. More details are described in Algorithm 1.

This algorithm guarantees that each chunk would only contain contiguous blocks inside the shuffle file up to a certain size, and blocks from different mappers belonging to the same shuffle partition would be pushed to the same Magnet shuffle service. To reduce the chances of blocks from different map tasks in the same shuffle partition getting pushed to the same Magnet shuffle service at the same time, each map task would randomize the order in which chunks are

Algorithm 1: Dividing blocks into chunks

Constants available to mapper:
number of shuffle partitions: R
max chunk size: L
offset array for shuffle blocks inside shuffle file: $l_1, l_2, \ldots, l_R$
shuffle services chosen by driver: $M_1, M_2, \ldots, M_n$

Variables:
current chunk to add blocks to: $C \leftarrow \{\}$
current chunk size: $l_c \leftarrow 0$
current shuffle service index: $k \leftarrow 1$

Output:
chunks and their associated Magnet shuffle services

```
for i = 1 ... R do
    if (i − 1)/(R/n) + 1 > k and k < n then
        output chunk and its shuffle service (C, M_k);
        C ← {block_i};
        l_c ← l_i;
        k++;
    else if l_c + l_i > L then
        output chunk and its shuffle service (C, M_k);
        C ← {block_i};
        l_c ← l_i;
    else
        C = C ∪ {block_i};
        l_c = l_c + l_i;
output chunk and its shuffle service (C, M_k);
```

processed. Once the chunks are divided and randomized, the map task would hand off to a dedicated thread pool to process transfers of these chunks and finish. Each chunk would be loaded from disk into memory, where the individual block inside the chunk gets pushed to the associated Magnet shuffle service. Note that this buffering of chunks happens inside Spark executors instead of Magnet shuffle services. More details on this are given in Section 4.1.

3.2.2 Merging Blocks on Magnet Shuffle Services

On the Magnet shuffle service side, for each shuffle partition from each Spark application that is actively being merged, the Magnet shuffle service would generate a merged shuffle file to append all received corresponding blocks. It also maintains some metadata for every actively merged shuffle partition. The metadata contains a bitmap tracking the mapper IDs of all merged shuffle blocks, a position offset indicating the offset in the merged shuffle file after the most recent successfully appended shuffle block, and a `currentMapId` that tracks the mapper ID of the shuffle block currently being appended. This metadata is uniquely keyed by the combination of the application ID, shuffle ID, and shuffle partition ID, and maintained as a `ConcurrentHashMap`. This is illustrated in Figure 5.

When Magnet shuffle service receives a block, it would retrieve the corresponding shuffle partition metadata before attempting to append the block into the corresponding merged shuffle file. The metadata can help Magnet shuffle service to properly handle several potential abnormal scenarios. The bitmap helps Magnet shuffle service to recognize any potential duplicate blocks, so no redundant data gets written into the merged shuffle file. In addition, even though the map tasks have already randomized the order of the chunks, a Magnet shuffle service may still receive multiple blocks from different map tasks belonging to the same shuffle

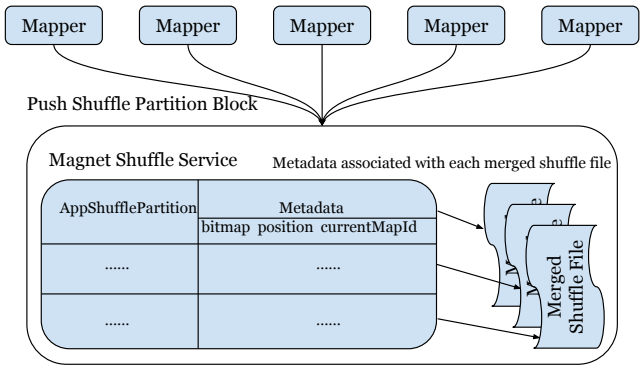


Figure 5: For each shuffle partition, Magnet shuffle service maintains a merged shuffle file and related metadata.

partition. When this happens, the `currentMapId` metadata can guarantee one block gets completely appended to the merged shuffle file before the next gets written to disk. Furthermore, it is possible that a block gets partially appended to the merged shuffle file before encountering a failure, which corrupts the entire merged shuffle file. When this happens, the position offset can help to bring the merged shuffle file back to a healthy state. The next block will be appended from the position offset, effectively overwriting the corrupted portion. If the corrupted block is the last block, the corrupted portion would be truncated when the block merge operation is finalized. By keeping track of this metadata, Magnet shuffle service can properly handle duplication, collision, and failure during the block merge operation.

3.3 Improving Shuffle Reliability

Magnet takes a best-effort approach and can fall back to fetching the original unmerged shuffle blocks. Thus any failures during the block push/merge operations is non-fatal:

- If the map task fails before materializing its shuffle data, no remote block push is triggered at this moment yet. Normal retry of the map task will initiate.
- If a shuffle block fails to be pushed to a remote Magnet shuffle service, after some reasonable retries, Magnet gives up on pushing this block and the associated chunk. These blocks that are not successfully pushed will be fetched in their original form.
- If Magnet shuffle service experiences any duplication, collision, or failure during the block merge operation, leading to blocks not getting merged, the original unmerged block will be fetched instead.
- If the reduce task fails to fetch a merged shuffle block, it can fall back to fetching the list of the unmerged shuffle blocks backing this merged shuffle block without incurring a shuffle fetch failure.

The metadata tracked in Magnet shuffle service, as introduced in Section 3.2.2, mostly improves the shuffle service's tolerance of merge failures. The additional metadata tracked in Spark driver helps to boost reduce task's fault-tolerance. As shown in Figure 4, in step 6, Spark driver would retrieve a list of `MergeStatus` from Magnet shuffle

services when it notifies them to stop merging blocks for a given shuffle. In addition, when each map task completes, it would also report a `MapStatus` to the Spark driver (step 5 in Figure 4). For a shuffle with M map tasks and R reduce tasks, the Spark driver would have gathered M `MapStatus` and R `MergeStatus` for this shuffle. Such metadata would tell Spark driver the location and size of every unmerged shuffle block and merged shuffle file, as well as which blocks get merged into each merged shuffle file. Thus, the Spark driver can build a complete picture about how to get the task inputs for each reduce task combining the merged shuffle file and unmerged blocks. When a reduce task fails to fetch a merged shuffle block, the metadata can also enable it to fall back to fetching the original unmerged blocks.

Magnet's best-effort approach effectively maintains 2 replicas of the shuffle data. While this helps to improve shuffle reliability, it also increases the storage need and write operations for the shuffle data. In practice, the former is not a major issue. The shuffle data is only temporarily stored on disks. Once a Spark application finishes, all of its shuffle data also gets deleted. Even though Spark applications shuffle petabytes of data daily in our clusters, the peak storage need for shuffle data is only hundreds of TB. The increased shuffle data storage need is a very small fraction of the total capacity in our clusters. We further discuss the implications of increased write operations in Section 4.3.

3.4 Flexible Deployment Strategy

In Sections 3.2-3.3, we have shown how Magnet integrates with Spark native shuffle by building on top of it. Different from [4, 33], Magnet allows Spark to natively manage all aspects of shuffle, including storing shuffle data, providing fault tolerance, and tracking shuffle data location metadata. Spark does not rely on external systems for shuffle in this case. This allows the flexibility to deploy Magnet in both on-prem clusters with co-located compute/storage nodes and cloud-based clusters with disaggregated storage layer.

For on-prem data centers that co-locate compute and storage nodes, data locality for shuffle reduce tasks can bring multiple benefits. This includes increased I/O efficiency and reduced shuffle fetch failures due to bypassing network transfers. By leveraging Spark's locality-aware task scheduling [12] and choosing Magnet shuffle services to push shuffle blocks based on locations of Spark executors, achieving shuffle data locality appears trivial. However, Spark's dynamic resource allocation [13] complicates this. The dynamic allocation feature allows Spark to release idle executors with no tasks running for a certain period of time and re-launch the executors later if tasks are pending again. This would make Spark applications more resource-efficient in a multi-tenancy cluster. This feature is enabled in LinkedIn's Spark deployment. Similarly, a few other large-scale Spark deployments also recommend this [10, 16].

With Spark dynamic allocation, when the driver selects the list of Magnet shuffle services at the beginning of a shuffle map stage (step 1 in Figure 4), the number of active Spark executors might be less than the desired number due to executor release at the end of the previous stage. If we choose Magnet shuffle services based on locations of Spark executors, we might end up with fewer shuffle services than needed. To address this issue, we choose Magnet shuffle services on locations beyond the active Spark executors, and launch Spark executors later via dynamic allocation based on locations of the chosen Magnet shuffle services. This way, instead of choosing Magnet shuffle services based on Spark executor locations, we launch Spark executors based on locations of Magnet shuffle service. Such an optimization is possible because of Magnet's integration with Spark native shuffle.

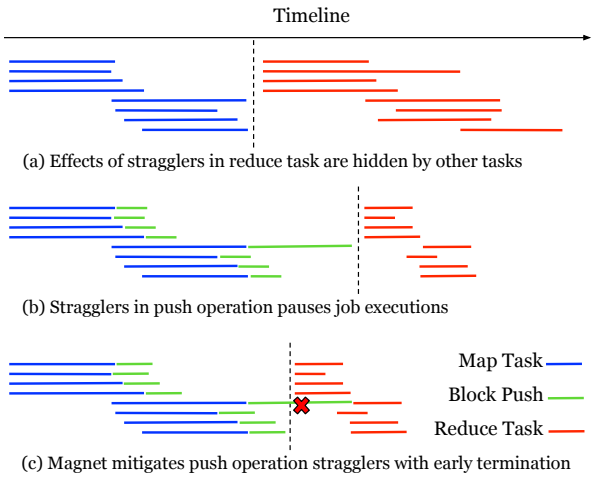


Figure 6: Handling stragglers in push operations.

For cloud-based cluster deployments, the compute and storage nodes are usually disaggregated. In such deployments, the intermediate shuffle data could be materialized in disaggregated storage via fast network connections [36, 15]. Data locality for shuffle reduce tasks no longer matters in such a setup. However, Magnet still fits well with such cloud-based deployments. Magnet shuffle services would run on the compute nodes, and store the merged shuffle files on the disaggregated storage nodes. By reading larger chunks of data instead of the small fragmented shuffle blocks over the network, Magnet helps to better utilize the available network bandwidth. Furthermore, the Spark driver can choose to optimize for better load balancing instead of data locality when selecting Magnet shuffle services. The Spark driver can query the load of the available Magnet shuffle services in order to pick ones that are less loaded. In our implementation of Magnet, we allow such flexible policies to select locations of Magnet shuffle services. So, we can choose to optimize for either data locality or load balancing or a combination of both based on the cluster deployment mode.

3.5 Handling Stragglers and Data Skews

Task stragglers and data skews are common issues in real-world data processing, and they slow down job executions. In order to become a practical solution in our environment, Magnet needs to be able to handle these potential issues.

3.5.1 Task Stragglers

When all the map tasks finish at the end of the shuffle map stage, the shuffle block push operation might not fully finish yet. There is a batch of map tasks which just started pushing blocks at this moment, and there could also be stragglers which cannot push blocks fast enough. Different from stragglers in the reduce tasks, any delay we experience at the end of the shuffle map stage will directly impact the job's runtime. This is illustrated in Figure 6. To mitigate such

stragglers, Magnet allows the Spark driver to put an upper bound on how long it is willing to wait for the block push/merge operation. At the end of the wait, the Spark driver would notify all the chosen Magnet shuffle services for the given shuffle to stop merging new shuffle blocks. This might lead to a few blocks not merged by the time the shuffle reduce stage starts, however it ensures Magnet can provide the majority of the benefit of push-merge shuffle while upper bounding the negative impact of stragglers to the Spark application's runtime.

3.5.2 *Data Skews*

Data skews happen when one or more shuffle partitions become significantly larger than the others. In Magnet, if we try to push and merge such partitions, we might end up merging a partition with 10s or even 100s GB of data, which is not desirable. There are existing solutions that deal with data skews in Spark, such as the adaptive execution feature [14]. With Spark adaptive execution, the statistics about each shuffle partition's size is gathered at runtime which can be used to detect data skews. If such a skew is detected, the operator triggering the shuffle, such as joins and group-bys, might divide the skewed partitions into multiple buckets in order to spread the computation of the skewed partition onto multiple reduce tasks. Magnet integrates well with such a skew-mitigating solution. When dividing the map task's shuffle blocks into chunks following Algorithm 1, Magnet can amend the algorithm by not including shuffle blocks larger than a predefined threshold into any chunks. This way, Magnet will merge all the normal partitions, but skip the skewed partitions as they would go beyond the size threshold. With Spark adaptive execution, the non-skewed partitions would still be fetched in their entirety, which can benefit from Magnet's push-merge shuffle. For the skewed partitions, since Magnet does not merge them and the original unmerged blocks are still available, it would not interfere with Spark adaptive execution's skew-mitigating solution.

4. IMPLEMENTATION OPTIMIZATIONS

We implemented Magnet on top of Apache Spark 2.3. Since the modifications are specific to Spark internals and not to the user-facing public APIs, the existing Spark applications can benefit from Magnet with no code changes. In order to achieve optimal efficiency and scalability, we also applied a few optimizations in our implementation.

4.1 Parallelizing Data Transfer and Task Execution

One of the areas that Spark is superior to the legacy Hadoop MapReduce engine is how it achieves better parallelization of shuffle data fetch and task execution. As illustrated in Figure 7(a), Hadoop MapReduce achieves limited parallelization via a technique termed "slow start". It would allow some reduce tasks to start fetching the shuffle data while some map tasks are still running. This way, only some reduce tasks' shuffle data fetch time overlaps with the map tasks executions. Spark does a much better job in this regard. Instead of overlapping map tasks and reduce tasks, which is not easy to manage in a DAG execution engine, Spark achieves the parallelization with asynchronous RPC. Separate threads are used for fetching remote shuffle blocks and executing reduce tasks on the fetched blocks. These 2 threads act like a pair of producers and consumers, allowing

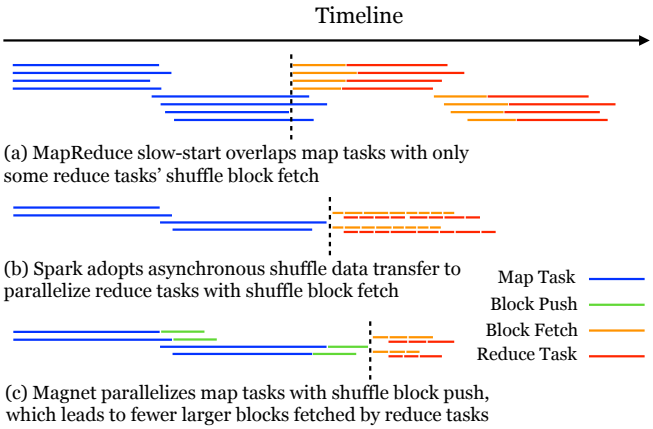


Figure 7: Magnet aims to parallelize task executions with shuffle data transfer.

better overlap between shuffle data fetch and reduce task executions. This is illustrated in Figure 7(b).

In our implementation of Magnet, we adopt a similar technique to achieve parallelized data transfer and task execution. On the shuffle map task side, by leveraging the dedicated thread pool, we decouple the shuffle block push operations from the map task executions. This is illustrated in Figure 7(c). This allows parallelized block push operations while later mappers are executing. While the shuffle map tasks might be more CPU/memory intensive, the block push operation would be more disk/network intensive. Such parallelization would help to better utilize the compute resources available to Spark executors.

On the shuffle reduce task side, if the merged shuffle file is located on a remote Magnet shuffle service, fetching the entire merged shuffle file as a single block might lead to undesired fetch delays, since we cannot achieve much parallelization between data fetch and reduce task execution. To address this issue, Magnet would divide each merged shuffle file into multiple MB-sized slices while blocks are appended to it. The slice boundary within the merged shuffle file is kept as a separate index file stored on disk. The Spark driver only needs to track information at the granularity of the merged shuffle partitions. How a merged shuffle file is further divided into slices is only tracked by the Magnet shuffle service via the index file. When the reduce task fetches a remote merged shuffle file, the Magnet shuffle service would respond with the number of slices so the client can convert a single fetch of the merged shuffle file into multiple fetches of the individual MB-sized slices. Such a technique helps to achieve a better balance between optimizing disk I/Os and parallelizing data transfer and task execution.

4.2 Resource Efficient Implementation

The implementation of the Magnet incurs very minimal CPU/memory overhead on the shuffle service side. Magnet shuffle service's only responsibility during the block push/merge operation is to accept remotely pushed blocks and append it to the corresponding merged shuffle file. This low overhead is achieved via the followings:

1. Different from [33], no sorting is performed on the shuffle service side. All sorting during the shuffle, either

map-side sort or reduce-side sort, are performed inside Spark executors by the map tasks and reduce tasks.

2. Different from [4, 41], Magnet shuffle service does not buffer shuffle blocks in memory during the merge operation. The only buffering of blocks happens inside Spark executors, and Magnet shuffle service merges blocks directly on disk.

The shuffle optimization in [41] is achieved via shuffle services pulling and merging shuffle blocks from local mappers. This requires buffering blocks in memory before merging to improve disk efficiency. The memory need grows as the number of concurrent merge streams in a shuffle service grows. In [41], it mentions that 6-8GB of memory is needed for 20 concurrent merge streams. This could become a scaling bottleneck in busy production clusters with much higher concurrency. Magnet buffers blocks in Spark executors instead, which distributes the memory needs across all executors. The limited concurrent tasks in Spark executors also keeps the memory footprint low, making Magnet more scalable. In [33, 4], the merge operation leverages per-partition write ahead buffers inside the shuffle service to batch write merged shuffle data. While this helps to reduce the number of write operations during merge, the memory requirement brings a similar scaling bottleneck as [41].

4.3 Optimizing Disk I/O

In Section 3.2, we showed how Magnet batch reads shuffle blocks to improve the disk I/O efficiency. This improvement from push-merge shuffle would require writing the intermediate shuffle data almost twice. Although this seems expensive, we argue that the overall I/O efficiency is still improved. Different from small random reads with HDDs, small random writes can benefit from multiple levels of caching such as the OS page cache and the disk buffer. These caches would group multiple write operations into one, reducing the number of write operations placed on the disk. As a result, small random writes can achieve a much higher throughput than small random reads. This is also shown in our benchmark in Section 5.2.3. With the batch reading of shuffle blocks and the caches batching the write operations, Magnet would reduce the overall disk I/O operations for shuffling small shuffle blocks even though it performs double write. For large shuffle blocks, as described in Section 3.5.2, we would skip merging these blocks. Thus the overhead of double write would not be incurred for them. In practice, because of Magnet's best-effort nature, we have also observed that choosing the appropriate OS I/O scheduler to prioritize read operations over writes can help to further reduce the overhead of double write. We are also looking into using small dedicated SSDs as a buffer to batch write even more blocks. Compared with the write ahead buffer approach in [33, 4], this improves the HDD write efficiency without incurring additional memory overhead.

5. EVALUATION RESULTS

In this section, we present benchmark results for Magnet. Via a combination of synthetic and production workload evaluations, we show that Magnet shuffle service can effectively improve the disk I/O efficiency and reduce the job runtime while being resource efficient.

	Size	# Map (# shuffle files)	# Reduce (# blocks per file)	Block
1	150 GB	1000	15000	10 KB
2	150 GB	1000	1500	100 KB
3	150 GB	1000	150	1000 KB

Table 1: Synthetic shuffle data configurations. It shows the total size of the data to be shuffled, the number of map tasks which equivalents to the number of shuffle files generated, the number of reduce tasks which equivalents to the number of blocks in each shuffle file, and the size of individual blocks.

5.1 Evaluation Setup

We evaluated Magnet under two different environments. The first is a distributed stress testing framework we developed to stress test the performance of a single Spark shuffle service. It can generate very high workloads for both shuffle block fetch and push operations in order to mimic the situation in a busy production cluster. More details on this stress testing framework are introduced in Section 5.2.1. The second is a benchmark cluster deployed with Magnet shuffle service. As introduced in Section 2.1, the Spark shuffle service runs within an underlying YARN NodeManager instance. In both environments, the YARN NodeManager instances are configured with only 3GB heap size. Furthermore, both environments use nodes with 56 CPU cores and 256 GB RAM each, connected with 10Gbps Ethernet links. For storage, each node has 6 HDDs, each with 100 MB/s I/O speed for sequential reads. A few nodes with SSDs are also used to evaluate the impact of a different storage medium.

5.2 Synthetic Workload Evaluations

We use a distributed stress testing framework for the synthetic workload evaluations. By stress testing against a single Spark shuffle service via simulating the workload it receives in a busy cluster during peak hours, we evaluate the completion time, disk I/O throughput and resource footprint of Magnet shuffle service against vanilla Spark.

5.2.1 Stress Testing Framework

To evaluate the characteristics of the Magnet shuffle service, especially under heavy shuffle workloads, we developed a distributed stress testing framework that allows us to simulate very high workloads a single Spark shuffle service would receive. Such a stress testing framework allows us to control the following 3 parameters, so that we can observe the performance of a Spark shuffle service even under situations that can only be found in a busy cluster during peak hours.

- Number of concurrent connections a single Spark shuffle service would receive
- Size of individual shuffle block size
- Total size of the intermediate shuffle data

Furthermore, this framework is able to stress test both the shuffle block fetch and push operations. The former would transfer shuffle blocks from a single Spark shuffle service to multiple clients, while the latter transfers shuffle blocks from multiple clients to a single shuffle service. With this

framework, we are able to evaluate both the vanilla Spark shuffle service and the Magnet shuffle service.

With this stress testing framework, we first generate synthetic shuffle data with certain block size and total size. Based on whether to evaluate the block fetch or push operation, such synthetic shuffle data could be generated on the one node running the Spark shuffle service, or distributed across multiple nodes running the clients. After generating the synthetic shuffle data, a certain number of clients are launched, each starting to fetch or push its portion of the shuffle blocks from/to the Spark shuffle service. These clients perform the same operation as the Spark map and reduce tasks in terms of how they fetch or push blocks. Each client could open multiple connections with the Spark shuffle service. The number of clients launched times the number of connections established per client becomes the number of concurrent connections the Spark shuffle service receives.

In our benchmark, we used 1 node to run the Spark shuffle service and 20 nodes to run the clients. We generated 3 different sets of synthetic shuffle data, as shown in Table 1. These 3 sets of synthetic shuffle data share the same total data size with varying block sizes.

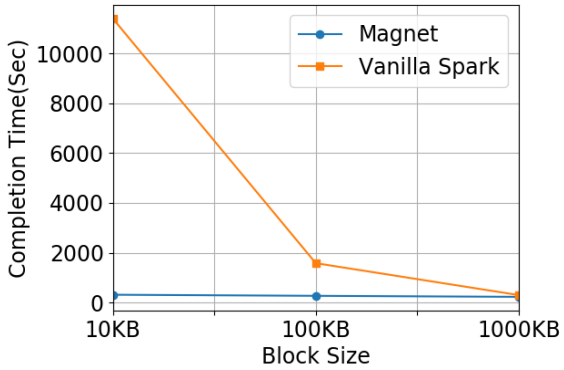
5.2.2 *Completion Time*

The first metric we want to evaluate is the completion time for transferring shuffle data. With Magnet, in order for the intermediate shuffle data to be transferred from map tasks to reduce tasks, it involves first pushing shuffle blocks to Magnet shuffle services for merging, then fetching the merged shuffle blocks from Magnet shuffle services for reduce task consumption. Here we evaluate the completion time for both the shuffle block fetch and push operations.

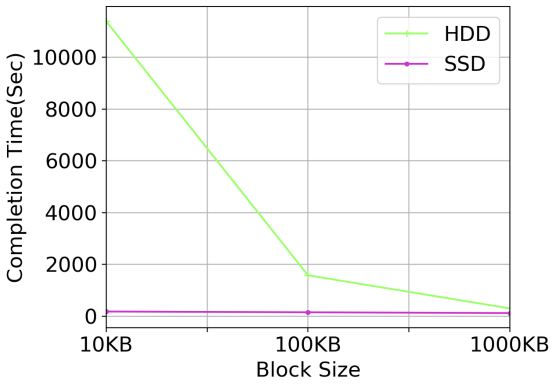
In this benchmark, we launched 3 runs for both the shuffle block fetch and push operations, with configurations as shown in Table 1. In addition, each run used 200 clients each with 5 connections. So the shuffle service received 1000 concurrent connections. As a comparison, the number of concurrent connections received by a Spark shuffle service in our production cluster during peak hours is around the same.

For the shuffle block fetch operation, as the block size grows, the I/O operations of the shuffle service gradually turns from small random read to large sequential read. This leads to better I/O efficiency and shorter completion time, as shown in Figure 8(a). As for the shuffle block push operation, since the clients would read large chunks of data from the shuffle file irrespective of the block size, the small block size would have less impact to the shuffle block push operation's efficiency. This is also shown in Figure 8(a). From this benchmark, we can clearly see that Magnet effectively mitigates the negative impact of small shuffle blocks when reading from HDDs. To transfer 150GB of 10 KB blocks with a single shuffle service, compared with vanilla Spark which takes 4 hours, Magnet would only take a little more than 5 minutes.

We further compare the time it takes to complete shuffle block fetch operation with HDDs vs. SSDs. Similar to the previous benchmark, 150 GB of data with varying block sizes is fetched from a single shuffle service using 1000 concurrent connections. As shown in Figure 8(b), SSDs show much more consistent performance as the block size changes. This further demonstrates that Magnet is able to achieve optimal disk efficiency for shuffle data transfer irrespective of the underlying storage hardware.



(a) Completion time for block push operation with Magnet and block fetch operation with vanilla Spark ESS



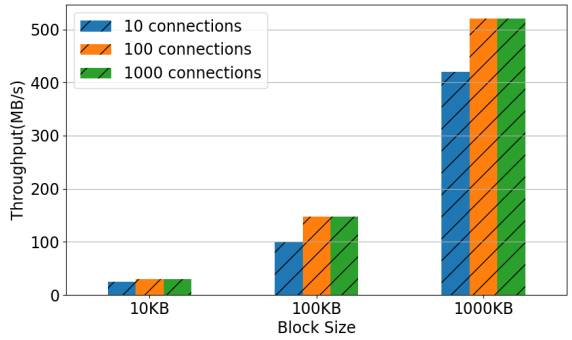
(b) Completion time for block fetch operation with vanilla Spark ESS using HDD vs. SSD

Figure 8: Completion time comparison for shuffle block push operation with Magnet shuffle service and shuffle block fetch operation with vanilla Spark ESS. (a) shows that the block push operation is not impacted much by small block size. (b) shows that block fetch operation requires storage mediums such as SSD to achieve the same. This validates Magnet can optimize shuffle data transfer irrespective of the underlying storage hardware.

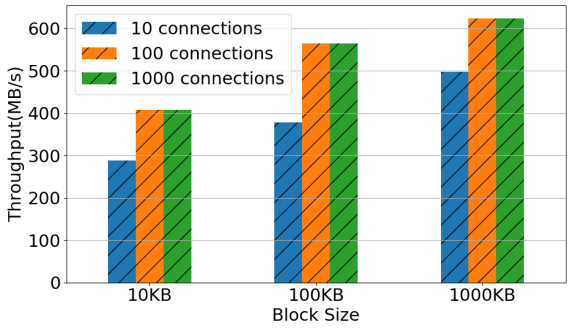
5.2.3 *Disk I/O*

We further evaluate the disk I/O efficiency with the shuffle block fetch and push operations. Since HDDs are subject to limited IOPS, we choose to measure the disk read/write throughput to compare how fast the shuffle service is able to read/write block data. When performing shuffle block fetch operations, the shuffle service is mainly reading shuffle blocks from disks. With shuffle block push operations, the shuffle service is mainly writing shuffle blocks to disks.

In this benchmark, in addition to evaluating with different block sizes, we also evaluate the impact of the number of concurrent connections the shuffle service receives. For block fetch operations, we observe that the shuffle service disk read throughput increases significantly as the block size increases (Figure 9(a)). This again illustrates the inefficiency of performing small reads with HDDs. In addition, due to the shuffle blocks being read only once in a random order, no caching would help to improve the read throughput. However, as the number of concurrent connections increases, we



(a) Disk read throughput of block fetch operation



(b) Disk write throughput of block push operation

Figure 9: Disk throughput comparison for shuffle block fetch and push operations. (a) shows that HDD read throughput with block fetch operations is severely impacted with small shuffle block size. (b) shows that HDD write throughput with block push operation is more consistent. In both cases, an increased number of connections does not lead to increased throughput, indicating the performance is bottlenecked at disk instead of network.

do not see any significant increase in the disk read throughput. This is the case for all 3 block sizes. In theory, an increased number of connections would mean more utilization of the available network bandwidth. The fact that we are not seeing any throughput increase indicates that the performance is bottlenecked at the disk and the clients are unable to saturate the available network bandwidth.

For block push operations, the disk write throughput is more consistent as the block size changes. When processing block push requests, Magnet shuffle service would append individual blocks to the merged shuffle file. Unlike the case for small random read, multiple levels of caching such as the OS page cache and the disk buffer would help to improve the performance of small random writes. Thus, the write throughput is much less impacted by the small block size. This is shown in Figure 9(b).

5.2.4 Resource Usage

We also evaluate the resource usage of Magnet shuffle services when performing the block push operations. The vanilla Spark shuffle services deployed in our production clusters can handle petabytes of daily shuffled data. This is largely due to the low resource usage of the shuffle service for both CPU and memory when performing the shuffle block fetch operations. For Magnet shuffle service to achieve the same, it also needs to have a low resource footprint.

In this benchmark, we evaluate both the shuffle fetch and push operations with 3 different block sizes as before. For each run, the shuffle service is receiving 1000 concurrent connections to simulate the scenario of shuffle services at peak hours. For CPU consumptions, Magnet shuffle service is showing a similar usage as the vanilla Spark shuffle service when data is being transferred. It averages between 20-50% of a CPU virtual core in the 3 runs with different block sizes. This is reasonable as the shuffle block push operation is a disk intensive operation, similar to the block fetch operation. For memory consumptions, Magnet shuffle service is also showing a similar footprint as the vanilla Spark shuffle service, averaging around 300 MB for both on-heap and off-heap memory. In addition, its memory usage is not impacted by the block size. This is large due to the implementation optimization as introduced in Section 4.2.

Table 2: Evaluation results with production workloads.

	Map Stage Runtime	Reduce Stage Runtime	Total Task Runtime
Workload 1 w/o Magnet	2.7 min	37 s	38 min
Workload 1 w/ Magnet	2.8 min	33 s	37 min
Workload 2 w/o Magnet	2.4 min	6.2 min	48.8 hr
Workload 2 w/ Magnet	2.8 min	2 min (-68%)	15 hr (-69%)
Workload 3 w/o Magnet	2.9 min	11 min	89.6 hr
Workload 3 w/ Magnet	4.1 min	2.1 min (-81%)	30.6 hr (-66%)
Workload 2+3 w/o Magnet	7.1 min	38 min	144.7 hr
Workload 2+3 w/ Magnet	8.5 min	6.3 min (-83%)	42.9 hr (-70%)

5.3 Production Workload Evaluations

We further evaluate Magnet with a few selected real workloads. This evaluation was done in a benchmark cluster with 200 nodes. We launched 200 Spark executors each having 5 GB memory and 2 vcores. We used 3 production jobs in our evaluation, representing small job (Workload 1), medium CPU-intensive job (Workload 2), large shuffle-heavy job (Workload 3). These workloads have the following characteristics:

- Workload 1 is a short SQL query which shuffles less than 100 GB data
- Workload 2 shuffles around 400 GB data, and is CPU intensive in both the shuffle map and reduce tasks
- Workload 3 is more I/O intensive, which shuffles around 800 GB data

We measured 3 metrics for these workloads with and without using Magnet shuffle service: 1) the end-to-end runtime of all shuffle map stages, 2) the end-to-end runtime of all reduce stages, and 3) total task runtime. The results are shown in Table 2. For Workload 1, since the shuffle overhead is low in this case, it would not benefit much from Magnet. The runtime comparison further validates that Magnet does not introduce much overhead leading to performance regression in this case. For Workload 2, the total job runtime saw a 45% reduction, and the reduce stage runtime and total task runtime saw a 68% reduction with Magnet. This validates that even for CPU intensive jobs, Magnet can provide significant improvement by optimizing the shuffle operation. For Workload 3, which is shuffle I/O intensive, Magnet helps to significantly decrease the job runtime. Specifically, Magnet reduced the shuffle reduce stage runtime by 81%, while keeping the shuffle map stage runtime largely unaffected.

In addition to measuring the performance of Magnet with a single workload, we also benchmarked with both Workload 2 and 3 running at the same time, each using 100 Spark executors. This is to measure the performance of Magnet with a mix of both CPU intensive and shuffle I/O intensive jobs, as well as when Magnet shuffle services serve both block push and fetch requests at the same time. From Table 2, we can see that the performance gains of Magnet is even more with combined workloads. This shows that while vanilla Spark shuffle service's performance degrades under increased shuffle workload, Magnet can continue to achieve superior shuffle performance.

6. RELATED WORK

Recently, there has been great research interest in optimizing shuffle for distributed data processing frameworks. Different from Magnet, where mappers push shuffle blocks to remote shuffle services to be merged, Riffle [41] performs block merge operation via shuffle services pulling shuffle blocks from local mappers. It has 3 disadvantages compared with Magnet: 1) Pull-based merge service requires buffering blocks in memory before merging to improve disk efficiency. The memory need grows as the number of concurrent merge streams in a shuffle service grows. This could become a scaling bottleneck in busy production clusters. Magnet buffers blocks in Spark executors instead, which distributes the memory needs across all executors. The limited concurrent tasks in Spark executors also keeps the memory footprint very low, making Magnet more scalable. 2) Local merge service does not relocate shuffle blocks, thus it cannot provide better shuffle data locality for reducers. 3) Similarly, local merge service does not replicate shuffle blocks on a different node, thus cannot help to improve fault-tolerance.

Sailfish [33]/Cosco [4] are two more solutions which merge shuffle blocks. Building on top of an abstraction of I-files, which is an extension to KFS [8], Sailfish aggregates Hadoop MapReduce shuffle data per partition into I-files, each consisting of multiple chunks. Cosco [4] is an implementation of Sailfish for Spark. Both solutions aim at disaggregated cluster deployments. The compute engines delegate to external storage systems, KFS for Sailfish and Facebook's disaggregated storage cluster for Cosco, to manage the shuffle intermediate data. This includes storing shuffle data, providing fault-tolerance, and tracking location metadata of shuffle blocks. On the other hand, Magnet integrates with Spark native shuffle, where Spark still manages all aspects of shuffle. Sailfish/Cosco's delegated shuffle approach has 2 disadvantages compared with Magnet: 1) The dependency on external storage systems makes it more restrictive to deploy. In cloud-based environments where dedicated storage layers already exist, it might not be possible to deploy custom storage solutions at scale. As LinkedIn is moving from on-prem clusters to Microsoft Azure [3], it is important to design a solution that is flexible to deploy in both environments. 2) With delegated shuffle, Spark cannot leverage the shuffle metadata hidden in external systems to get smart with scheduling reducers or handle shuffle fetch failures. Magnet's integration with Spark native shuffle helps to improve performance by offering better shuffle data locality. While the integrated shuffle approach provides a more flexible deployment strategy and the native shuffle performance, the delegated shuffle approach is less coupled with a specific engine and thus can more easily extend to support multiple compute engines.

iShuffle [25] is another work that optimizes Hadoop MapReduce shuffle via a shuffle-on-write technique. It would push mapper generated shuffle blocks to the reducers. Compared with Magnet, iShuffle is subject to the impact of stragglers and does not help to improve shuffle reliability. In addition, due to the limitations of Hadoop MapReduce, which adopts 1-level monolithic scheduling instead of Spark on YARN's 2-level scheduling mechanism [34], iShuffle also cannot effectively schedule reduce tasks to leverage data locality in a multi-tenancy cluster.

There are a few other studies which aim to improve shuffle with different approaches. HD shuffle [31] proposed a new shuffle algorithm that would divide a single large fan-in fan-out shuffle into multiple ones each with a bounded fan-in fan-out. While this can help to improve the disk efficiency, it introduces additional shuffles which might not be desirable. MapReduce Online [20] proposed a push-based mechanism to optimize online aggregation and continuous queries. It works for in-memory shuffle, and is not designed for large jobs. Hadoop-A [39] proposed an approach to optimize shuffle leveraging RDMA, and Splash from MemVerge [42] is a solution that leverages PMEM. The benefits of these solutions are coupled with non-commodity hardware. Worth noting is that Riffle [41], Sailfish [33], and Cosco [4] are the only known solutions so far that are deployed in production at scale.

7. CONCLUSIONS

In this paper, we present Magnet, a Spark shuffle service leveraging push-merge shuffle to improve the efficiency, reliability, and scalability of the shuffle operation in Spark. Magnet achieves optimized disk I/Os on the shuffle write and read paths via merging shuffle blocks. It further reduces shuffle related failures by replicating the shuffle data in both the original and the optimized form. Magnet also improves data locality for shuffle reduce tasks which further improves the efficiency and reliability of shuffle in Spark. Based on our evaluations, we show that Magnet helps to mitigate several existing issues with Spark shuffle operation and leads to shorter job completion time.

8. REFERENCES

[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. Gonçalves, J. Pereira, and R. Jiménez-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. *Proceedings of the VLDB Endowment*, 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.