Magnet: A scalable and performant shuffle architecture for Apache Spark

Min Shen October 21, 2020

Co-authors: Min Shen, Chandni Singh, Ye Zhou, and Sunitha Beeram

At LinkedIn, we rely heavily on offline data analytics for data-driven decision making. Over the years, Apache Spark has become the primary compute engine at LinkedIn to satisfy such data needs. With its unique features, Spark empowers many business-critical tasks at LinkedIn, including data warehousing, data science, AI/ML, A/B testing, and metrics reporting. The number of use cases requiring large scale data analytics is also growing very fast. From 2017 till now, Spark usage at LinkedIn has grown about 3X year over year. As a result, the Spark engine at LinkedIn now operates on top of a massive infrastructure. With more than 10,000 nodes across our production clusters, Spark jobs now account for more than 70% of cluster compute resource usage and tens of PB of daily processed data. Tackling scaling challenges to ensure our Spark compute infrastructure grows sustainably is at the core of what LinkedIn's Spark team does.

Although Apache Spark has many benefits that contribute to its popularity at LinkedIn and among the industry in general, we've still experienced several challenges in operating Spark at our scale. As outlined in our Spark + AI Summit 2020 talk, these challenges cover multiple layers in the stack, including compute resource management, compute engine scalability, and user productivity. We will focus on Spark shuffle scalability challenges in this blog post and introduce Magnet, a novel push-based shuffle service.

Shuffle basics

Data shuffle is a vital operation in the MapReduce compute paradigm powering Apache Spark and many other modern big data compute engines. The shuffle operation basically transfers intermediate data via all-to-all connections between the map and reduce tasks of the corresponding stages. Through shuffle, the data is properly partitioned across all the shuffle partitions, according to the value in each record's partition key. This is illustrated in Figure 1:

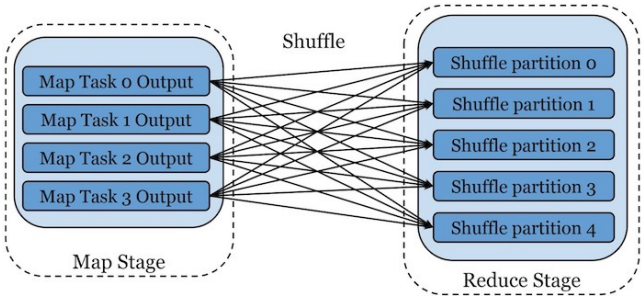

Figure 1: Basic shuffle operation

While the basic concept of the shuffle operation is straightforward, different compute engines have taken different approaches to implementing it. At LinkedIn, we run Spark on top of Apache YARN, and leverage Spark's External Shuffle Service (ESS) to operate its shuffle. This is illustrated in Figure 2:

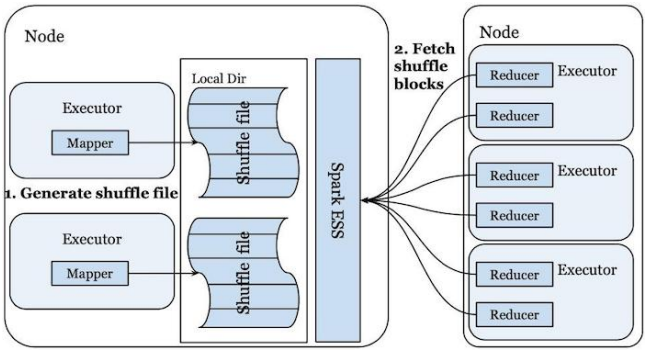

Figure 2: Spark's shuffle operation with External Shuffle Service

With this setup, each node in the compute cluster is deployed with one Spark ESS instance. When the Spark executors run the mapper tasks of a shuffle, these tasks will generate shuffle files onto local disks. Each shuffle file consists of multiple shuffle blocks, each representing the data belonging to the corresponding shuffle partition. After these shuffle files are generated, the local Spark ESS instances know where to locate these shuffle files and the individual shuffle blocks generated by different mapper tasks. When the Spark executors start running the reducer tasks of the same shuffle, these tasks will get the information from the Spark driver about where to find the shuffle blocks as their task inputs. These reducers will then send requests to remote Spark ESS instances, trying to fetch their corresponding shuffle blocks. The Spark ESS, upon receiving such requests, will read the corresponding shuffle blocks from local disks and send the data back to the reducers.

Challenges

Spark's existing shuffle mechanism achieves a good balance between performance and fault-tolerance requirements. However, when we operate Spark shuffle at our scale, we have experienced multiple challenges, which have made the shuffle operation a scaling and performance bottleneck in our infrastructure.

Reliability issue

The first challenge is a reliability issue. In our production clusters, due to the sheer volume of compute nodes and the scale of the shuffle workload, we have noticed shuffle service availability problems during cluster peak hours. This can result in a shuffle fetch failure, leading to expensive stage retries, which can be pretty disruptive because they cause workflow SLA violations and job failures. This is further illustrated in Figure 3, which shows the number of daily Spark stage failures due to shuffle. This was trending very high, at hundreds to more than one thousand per day by the end of 2019.

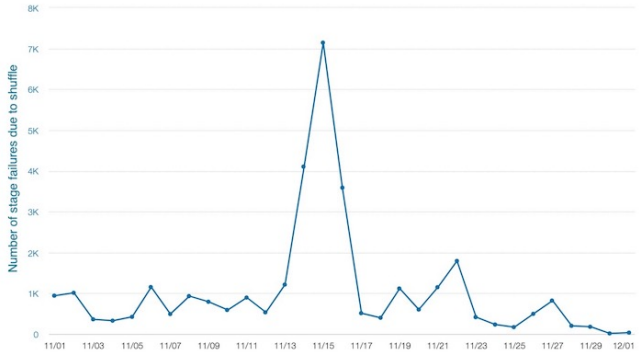

Figure 3: Daily number of shuffle fetch failures

Efficiency issue

The second challenge is an efficiency issue. At LinkedIn, we store shuffle files on HDDs. Since the reducers' shuffle fetch requests arrive in random order, the shuffle service also accesses the data in the shuffle files randomly. If the individual shuffle block size is small, then the small random reads generated by shuffle services can severely impact the disk throughput, extending the shuffle fetch wait time. This is also illustrated in Figure 4. As this dashboard shows, between March and August in 2020, somewhere between 10-20% of our production clusters' compute resources were wasted on shuffle, sitting idle while waiting for remote shuffle data.

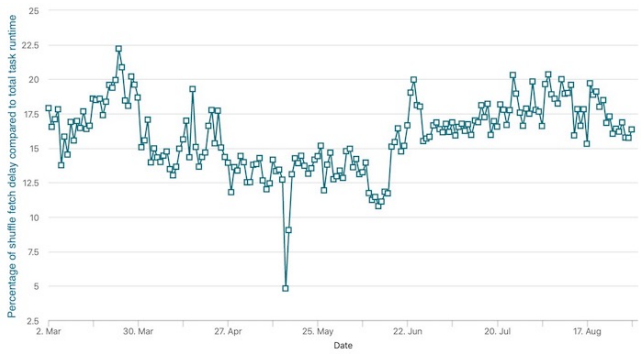

Figure 4: Daily average shuffle fetch delay as a percentage of total compute time

Scalability issue

The third challenge is a scaling issue. Since the external shuffle service is a shared service in our infrastructure, an incorrectly-tuned job stressing a few shuffle services can affect other jobs as well. When an incorrectly-tuned job that's shuffling many small shuffle blocks stresses a shuffle service, it brings performance degradation to

not only itself, but also to all the neighboring jobs sharing the same shuffle service. This can lead to unpredictable runtime delays for jobs that are otherwise behaving normally, especially during cluster peak hours.

Magnet shuffle service

To address these issues, we have designed and implemented Magnet, a novel push-based shuffle service. Project Magnet made its debut to the community earlier this year as an industrial track paper published in VLDB 2020, and you can read our VLDB paper here: Magnet: Push-based Shuffle Service for Large-scale Data Processing. More recently, we are in the process of contributing Magnet back to Apache Spark. The remainder of this blog post will cover the high-level designs behind Magnet and its performance in production, but interested readers can watch the SPIP Jira for updates on that effort and the SPIP document for implementation level details.

Push-based shuffle

The core idea behind Magnet shuffle service is the concept of push-based shuffle, where the mapper-generated shuffle blocks also get pushed to remote shuffle services to be merged per shuffle partition. The shuffle write path of push-based shuffle is illustrated in Figure 5:

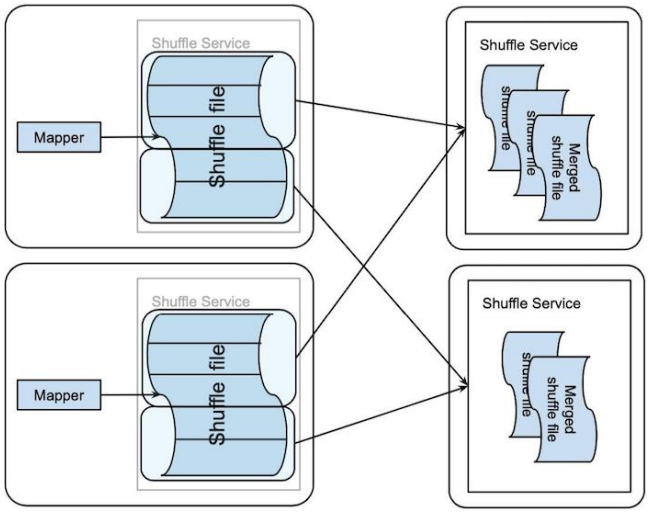

Figure 5: Shuffle write path of push-based shuffle

After the map task generates its shuffle file, it prepares the shuffle blocks to be pushed to remote ESSs. It groups the contiguous blocks in the shuffle file into MB-sized chunks, and identifies the corresponding ESS to push this group to. Shuffle blocks that are larger than a certain size are skipped, so we do not push blocks that are potentially coming from large skewed partitions. The map task determines this grouping and the corresponding ESS destinations in a consistent way such that blocks from different mappers belonging to the same shuffle partition are pushed to the same ESS. Once the grouping is done, the transfer of these blocks is handed off to a dedicated thread pool and the map task simply finishes. This way, we decouple the task execution threads from the block transfer threads, achieving better parallelism between the I/O intensive data transfer and the CPU intensive task execution. The ESS accepts remotely pushed shuffle blocks and merges them into the corresponding merged shuffle file for each unique shuffle partition. This is done in a best-effort manner, which does not guarantee all blocks get merged. However, the ESS does guarantee that no data duplication or corruption happens during the merge.

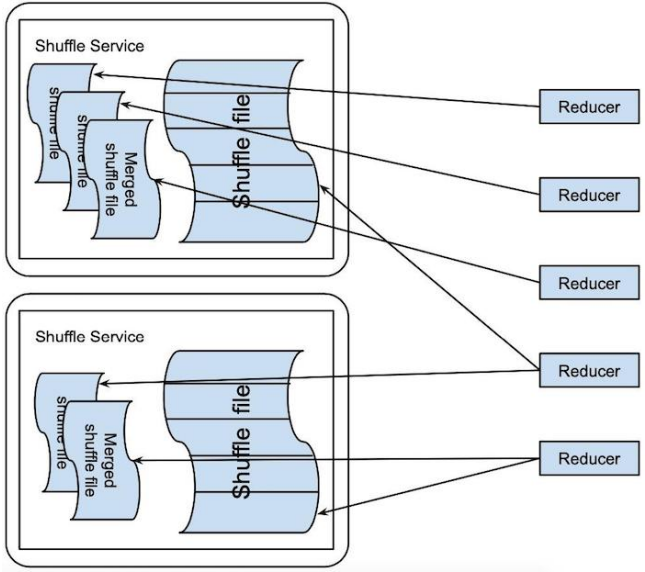

Figure 6: Shuffle read path of push-based shuffle

On the shuffle read path of push-based shuffle, the reduce tasks can fetch their task inputs from both the merged shuffle files and the original shuffle files generated by the map tasks (Figure 6). The ESS can perform large sequential I/O instead of small random I/O when reading from the merged shuffle files, significantly increasing the I/O efficiency. Taking advantage of this, the reduce tasks would prefer to fetch their inputs from the merged shuffle files. Since the block push/merge process is best-effort, the reduce tasks can use the unmerged blocks to fill any holes in the merged shuffle files. They can even completely fall back to fetching the unmerged blocks in case the merged shuffle files become unavailable. The efficient disk I/O pattern of Magnet further provides more flexibility for building a performant Spark cluster, as it's much less reliant on SSDs for achieving good shuffle performance.

The Spark driver is responsible for coordinating push-based shuffle across the map and reduce tasks. On the shuffle write path, the Spark driver determines a list of ESSs for the map tasks of a given shuffle to work with. This list of ESSs is sent to the Spark executors as part of the task context, which enables the map tasks to come up with the above mentioned consistent mapping between block groups and remote ESS destinations. The Spark driver further coordinates the transition between the map and reduce stage. Once all the map tasks are finished, the Spark driver waits for a configurable amount of time before notifying all the chosen ESSs for this shuffle to finalize the merge operation. When an ESS receives a finalize request, it stops accepting any new blocks from the given shuffle. It also responds to the driver with a list of metadata for each finalized shuffle partition, which includes location and size information about the merged shuffle files as well as a bitmap indicating which blocks have been merged. Once the Spark driver receives such metadata from all ESSs, it starts the reduce stage. At this point, the Spark driver has the complete view of the locations of the shuffle data, which is now 2-replicated between the merged shuffle files and the original shuffle files. The Spark driver leverages this information to coordinate the reduce tasks for their input locations. Furthermore, the locations of the merged shuffle files create a natural locality preference for the reduce tasks. The Spark driver leverages that information to do locality-aware scheduling of reduce tasks in clusters that have collocated shuffle storage. This is illustrated below in Figure 7:

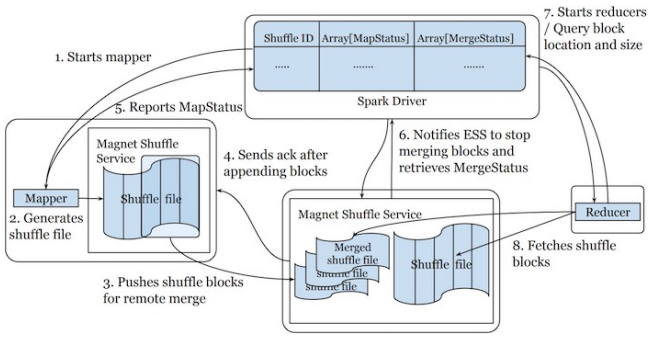

Figure 7: Overall architecture of push-based shuffle

Benefits of push-based shuffle

Push-based shuffle brings several key benefits to the Spark shuffle operation

Improved disk I/O efficiency

With push-based shuffle, the shuffle service switches from small random reads to large sequential reads when accessing shuffle data in the shuffle files, which significantly improves disk I/O efficiency, especially for HDD-based shuffle storage. On the shuffle write path, even though the shuffle data is written twice for small blocks, the overall I/O efficiency is still improved. This is because small random writes can benefit from multiple levels of caching, such as the OS page cache and the disk buffer. As a result, small random writes can achieve much higher throughput than small random reads. The effect of the improved disk I/O efficiency is reflected in the performance numbers shown later in this blog post. More detailed analysis on the I/O efficiency improvement is included in our VLDB paper.

Mitigate shuffle reliability/scalability issues

For a Spark vanilla shuffle operation to succeed, it requires every reduce task to successfully fetch every corresponding shuffle block from all map tasks, which often cannot be satisfied in a busy cluster with thousands of nodes. Magnet achieves a better shuffle reliability in multiple ways:

1. Magnet adopts a best-effort approach for pushing blocks. Failures during the block push/merge process do not impact the shuffle process.
2. With push-based shuffle, Magnet effectively generates a second replica of the shuffle intermediate data. A shuffle fetch failure can only happen if a shuffle block cannot be fetched from either the original shuffle file or the merged shuffle file.

With the locality-aware scheduling of the reduce tasks, they are often collocated with the corresponding merged shuffle files, allowing them to read the shuffle data bypassing ESS. This makes the reduce tasks much more resilient to ESS availability or performance issues, thus alleviating the scalability issue mentioned previously.

Handle stragglers during block push process

In Spark's vanilla shuffle operation, the effects of stragglers in tasks (i.e., a few tasks running significantly slower than others) can be hidden by other tasks, since multiple tasks are usually running concurrently. With push-based shuffle, if there were any stragglers in the block push operation, it could potentially pause job executions for a long time. This is because the block push operation is in between the shuffle map and reduce stage. When stragglers are present, there might be no tasks running at all. Magnet can effectively handle stragglers during the block push process, however, via the early termination technique. Instead of waiting for the push process to completely finish, Magnet limits the time it waits between the shuffle map and reduce stages. Magnet's best-effort nature makes it tolerant of the unmerged blocks due to early termination. This is illustrated below in Figure 8:

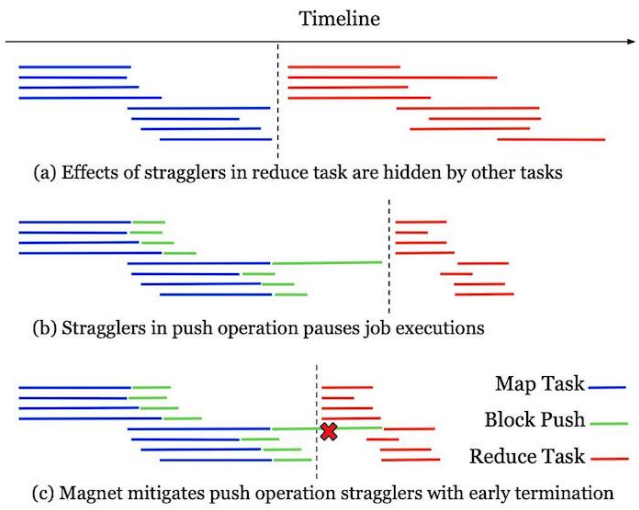

Figure 8: Magnet can mitigate push operation stragglers

Native integration with Spark

Magnet integrates natively with Spark, and this brings multiple benefits:

1. Magnet has no dependencies on other external systems. This helps to simplify the deployment, monitoring, and productionizing of Magnet shuffle service.
2. Through native integration with Spark's shuffle system, the metadata in Magnet shuffle service is exposed to the Spark driver as well. This enables the Spark driver to achieve better performance (via locality-aware scheduling of tasks) and better fault tolerance (via fallback to original shuffle blocks).
3. Magnet works well with existing Spark features such as Adaptive Query Execution. One of the promises in Spark AQE is the ability to dynamically optimize skew join, which also requires special handling on shuffle. Spark AQE

would divide a skewed shuffle partition among multiple reducer tasks, each fetching shuffle blocks from only a sub-range of mapper tasks. Since the merged shuffle file no longer maintains the original boundary of each individual shuffle block, it would be impossible to divide a merged shuffle file in the way required by Spark AQE. Since Magnet keeps both the original shuffle files and the merged shuffle files instead, it can delegate to AQE to handle skewed partitions while optimizing shuffle operations for the non-skewed partitions.

Performance comparison

We have evaluated the performance of Magnet using real production jobs at LinkedIn, and we have seen very promising results. In the table below, we show the performance results of running one ML feature generation job that has tens of shuffle stages and close to 2 TB of shuffle data. Magnet achieves very good performance results compared with the vanilla shuffle in Spark. Note that Magnet brings a 98% reduction for the shuffle fetch wait time. This is possible through Magnet's efficient shuffle disk I/O and locality-aware scheduling of reduce tasks. In addition, this production job is not entirely using Spark SQL, as it uses a mixture of the declarative and imperative side of Spark to compose its computation logic. When optimizing shuffle operations, Magnet assumes very little of the job itself.

	Total shuffle fetch wait time (min)	**Total executor task runtime (min)**	**End-to-end job runtime (min)**
Vanilla Spark shuffle	20636	50771	42
Magnet shuffle	445 (-98%)	29928 (-41%)	31 (-26%)

We have also onboarded more production flows at LinkedIn that are shuffle heavy. An estimated 15% of the shuffle workload in one of our production clusters has been migrated to Magnet. Across these jobs, we have seen similar reductions in shuffle fetch wait time, task total runtime, and job end-to-end runtime. As illustrated in Figure 9, the Magnet-enabled Spark jobs see an average of 3-4X reduction in shuffle fetch wait time. Furthermore, we have seen around 10X increase in the amount of shuffle data that is accessed locally, indicating the much-improved data locality with push-based shuffle. This is illustrated in Figure 10. Last but not least, we have seen that the job runtime becomes more stable during cluster peak hours. As we onboard more flows, Magnet converts more shuffle workload into the optimized path, reducing pressures on shuffle services and bringing even more benefits to the cluster. On the other hand, Magnet could potentially double the shuffle temporary storage need. We are mitigating this by switching to the zstd compression codec for shuffle files, which could potentially reduce the shuffle file size by 50% compared with the default compression codec.

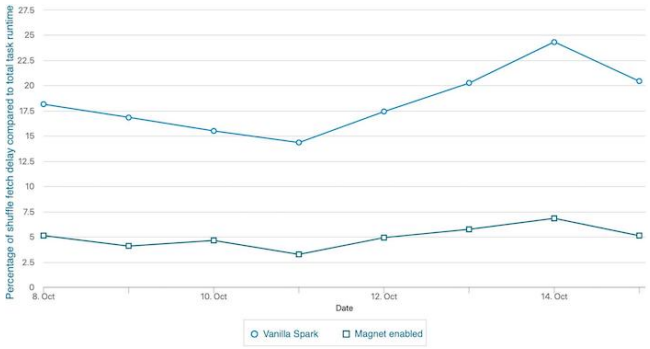

Figure 9: Daily average shuffle fetch delay between Magnet-enabled and vanilla jobs

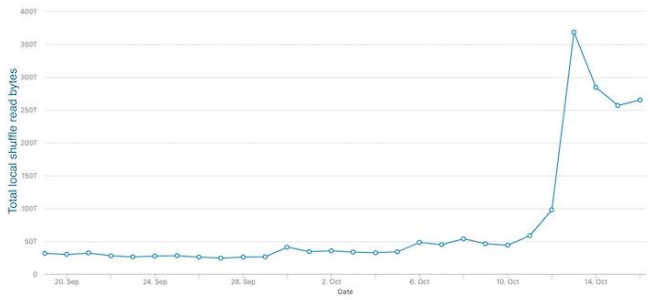

Figure 10: Increase of local shuffle read data size with Magnet-enabled jobs

Conclusion and future work

In this blog post, we have introduced Magnet shuffle service, a next-gen shuffle architecture for Apache Spark. Magnet improves the overall efficiency, reliability, and scalability of the shuffle operation in Spark. Recently, we have also seen other

solutions proposed in the industry that target the shuffle process, specifically Cosco, Riffle, Zeus, and Sailfish. We have made a comparison between Magnet and these other solutions, especially Cosco, Riffle, and Sailfish, in our VLDB paper.

In the future, we are also considering making Magnet push-based shuffle available in other deployment environments and compute engines. Our current cluster is deployed on-prem as a compute/storage collocated cluster. As LinkedIn is migrating towards Azure, we are also evaluating ways to adapt push-based shuffle for compute/storage disaggregated clusters. In addition, our current design for push-based shuffle is mostly targeting batch engines, and we are also considering its applicability to streaming engines as well.

Acknowledgements

It takes a dedicated team to bring a project of the magnitude of Magnet to see the light of day. In addition to efforts from Min Shen, Ye Zhou, and Chandni Singh, the project has been significantly contributed to by Venkata Krishnan Sowrirajan and Mridul Muralidharan. Erik Krogen, Ron Hu, Minchu Yang, and Zoe Lin have contributed to production rollout and observability improvements around Magnet. Special shoutout to Yuval Degani for building GridBench—this tool has made it very easy to understand the impact of various factors on job runtime. Special thanks to our partner teams, especially Jan Bob and Qun Li's team, for being early adopters of Magnet.

Large infrastructure efforts like Magnet require significant and sustained commitment from management. Sunitha Beeram, Zhe Zhang, Vasanth Rajamani, Eric Baldeschwieler, Kapil Surlakar, and Igor Perisic: thank you for your unyielding support and guidance. Magnet's design has also benefited from reviews and deep discussions with Sriram Rao and Shirshanka Das.

Magnet has received tremendous support from the open source Apache Spark community. We are grateful for partnership with Databricks and for the reviews from numerous community members.

Topics

Spark, infrastructure, Data, Open Source