Support push-based shuffle to improve shuffle efficiencys

Q1. What are you trying to do? Articulate your objectives using absolutely no jargon.

In a large deployment of a Spark compute infrastructure, Spark shuffle is becoming a potential scaling bottleneck and a source of inefficiency in the cluster. When doing Spark on YARN for a large-scale deployment, people usually enable Spark external shuffle service and store the intermediate shuffle files on HDD. Because the number of blocks generated for a particular shuffle grows quadratically compared to the size of shuffled data (# mappers and reducers grows linearly with the size of shuffled data, but # blocks is # mappers * # reducers), one general trend we have observed is that the more data a Spark application processes, the smaller the block size becomes. In a few production clusters we have seen, the average shuffle block size is only 10s of KBs. Because of the inefficiency of performing random reads on HDD for small amount of data, the overall efficiency of the Spark external shuffle services serving the shuffle blocks degrades as we see an increasing # of Spark applications processing an increasing amount of data. In addition, because Spark external shuffle service is a shared service in a multi-tenancy cluster, the inefficiency with one Spark application could propagate to other applications as well.

There are a few prior works that aim at solving this problem with different approaches. Facebook gave 2 talks at Spark+AI Summit in the past 2 years (1, 2). Their solutions aim at enabling merging smaller shuffle blocks into larger ones to improve the disk efficiency. In addition, we have seen solutions from a few storage vendors such as Alluxio and MemVerge.

There is also a community effort in SPARK-25299 which is to define an API that allows using different external storage for storing the shuffle data. This category of solutions enables leveraging storage mediums more optimized for such small random reads to serve the shuffle blocks.

Due to the constraints of large deployments of Spark compute infrastructure, it might not be practical/feasible to store all the intermediate shuffle files in a more optimized storage medium. In this SPIP, we propose a Spark-native approach to enhance its shuffle mechanism to combat these identified scaling and efficiency bottlenecks in a large-scale Spark deployment. Instead of reducers fetching the individual small shuffle blocks, which leads to many of the issues we have seen, we enable pushing shuffle blocks to remote shuffle services after being generated by the mappers to allow pre-merging them into larger ones before fetched by the reducers (refer to Appendix A/B for more details). The goals of this SPIP are thus:

1. Enhance the efficiency of the shuffle process in Spark when dealing with large shuffles (large in terms of shuffle data sizes and/or block counts)
2. Still bring benefits or at least introduce close-to-zero overhead when dealing with smaller shuffles

Q2. What problem is this proposal NOT designed to solve?

This SPIP is not targeting enabling storing shuffle data in disaggregated storage, which is being addressed by SPARK-25299.

The initial version of this SPIP can only work when Spark Adaptive Execution is turned off (so each reducer will fetch corresponding shuffle partition blocks from ***all*** mappers). We can target making this solution working with Spark Adaptive Execution later.

Q3. How is it done today, and what are the limits of current practice?

We have seen 2 categories of solutions dealing with this problem as described in Q1. One category of the solutions is to leverage storage mediums (such as memory or SSD) that are more optimized for such small random reads for serving shuffle blocks. In a large-scale deployment, the amount of shuffle data stored concurrently could be 100s of TBs or even reaching PBs, which will be fetched by 10s or 100s of thousands of Spark executors spread between thousands of compute nodes. Offloading all the intermediate shuffle data storage to such optimized storage medium might not be practical from various aspects.

Another category of the solutions is to change the behavior of the shuffle process in Spark to merge smaller shuffle blocks into larger ones to improve the efficiency. We have seen Facebook's solutions in prior Spark+AI Summit talks. This problem is also a well-researched area. There are multiple published literatures studying this problem, such as MapReduce online, Sailfish, Riffle, ishuffle etc. From SPARK-2044, it appears that even the Spark community investigated doing this in Spark at some time. However, none of these solutions are available for Spark in an open-source form.

Q4. What is new in your approach and why do you think it will be successful?

We propose a solution leveraging push-based shuffle where shuffle is performed after being written by mappers and blocks get pre-merged and move towards reducers. We take a Spark-native approach to achieve this, i.e., extending Spark's existing shuffle netty protocol, and the behaviors of Spark mappers, reducers and drivers. This way, we can bring the benefits of more efficient shuffle in Spark without incurring the dependency or overhead of either specialized storage layer or external infrastructure pieces.

We have already developed a prototype based on top of a patched version of Spark 2.3, and have seen significant efficiency improvements when performing large shuffles.

Q5. Who cares? If you are successful, what difference will it make?

Spark application developers who can expect speedup to their Spark applications.

Cluster admins who can expect more scalable and more optimized infrastructure.

Q6. What are the risks?

Push-based shuffle brings the most benefit for a large shuffle. For a small shuffle, e.g. ones that shuffle less than 1 GB of data, we won't see much improvement from push-based shuffle. In this case, should we still do push-based shuffle? For now, we have been thinking of introducing new configurations to determine whether to do push-based shuffle or not for a given shuffle based on # mappers/reducers or shuffle data size. By allowing cluster admins to configure these settings, we can enable them to optimize for the right type of shuffles in the cluster. However, if a cluster's workload is dominated by small shuffles, we might be skipping too many shuffles.

With push-based shuffle, the Spark driver needs to track additional information in its MapOutputTracker. We will need to extend the data structure in MOT to track metadata about the merged blocks. Would the potentially increased driver memory needs be a concern?

Spark Adaptive Execution would allow a reducer to fetch blocks from multiple shuffle partitions or only from a subset of all the mappers in order to dynamically change the parallelism in the reducer phase. For the latter case, where a reducer is not fetching blocks from all mappers for a given partition, we cannot enable push-based shuffle yet since we are right now merging blocks from all mappers for a given partition. In case of data skews, where we could see a large shuffle block generated by a mapper, we can skip pushing these blocks if their size goes beyond a configured threshold (more details in Appendices). However, this is not sufficient to make it work with Adaptive Execution yet.

Q7. How long will it take?

Multiple months due to the complexity of this problem and the necessity of achieving consensus with the community on the design.

Q8. What are the mid-term and final "exams" to check for success?

Success is for push-based shuffle to become an optional mechanism in Spark that complements sort-based shuffle to improve the overall shuffle efficiency. Mid-term might be to put in the changes for shuffle netty protocol, while the behavior changes for mapper (shuffle writer), reducer (shuffle reader), and driver are still being worked on.

Appendix A. Proposed API Changes. Optional section defining APIs changes, if any. Backward and forward compatibility must be taken into account.

No user-facing Spark API is changed in this design. There are however Spark internal APIs that need to be extended as part of this design.

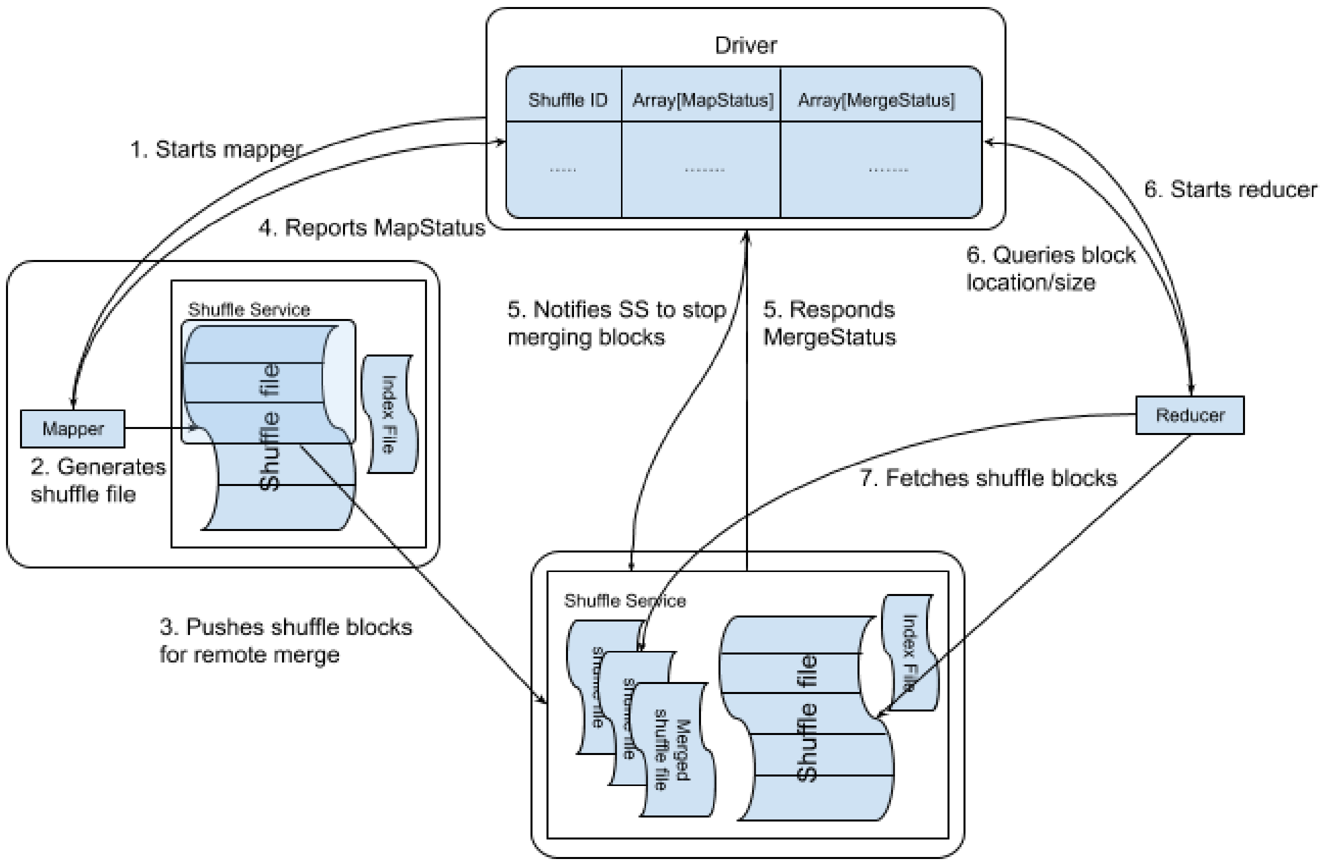
On the shuffle netty protocol side, we are leveraging SPARK-6237 to provide Spark external shuffle services the ability to receive remotely pushed blocks as a stream of data, without having to buffer the entire message in the FrameDecoder. Similar to SPARK-24296, we are extending Spark external shuffle service's RPC handler to handle a stream of remotely pushed blocks.

On the shuffle client side, we are extending ExternalShuffleClient to add additional APIs for data plane and control plane RPCs.

On the driver side, specifically inside ShuffleStatus within MapOutputTracker, for each shuffle we are optionally also tracking an array of MergeStatus. MergeStatus is similar to MapStatus, but instead of tracking metadata of map generated blocks, it tracks the metadata (location, size, etc) for a merged block. With this extension, for any shuffle which is leveraging push-based shuffle, an array of # mappers MapStatus and an array of # reducers MergeStatus will be tracked inside ShuffleStatus.

Appendix B. Optional Design Sketch: How are the goals going to be accomplished? Give sufficient technical detail to allow a contributor to judge whether it's likely to be feasible. Note that this is not a full design document.

An overview of the shuffle steps with push-based shuffle is described below:

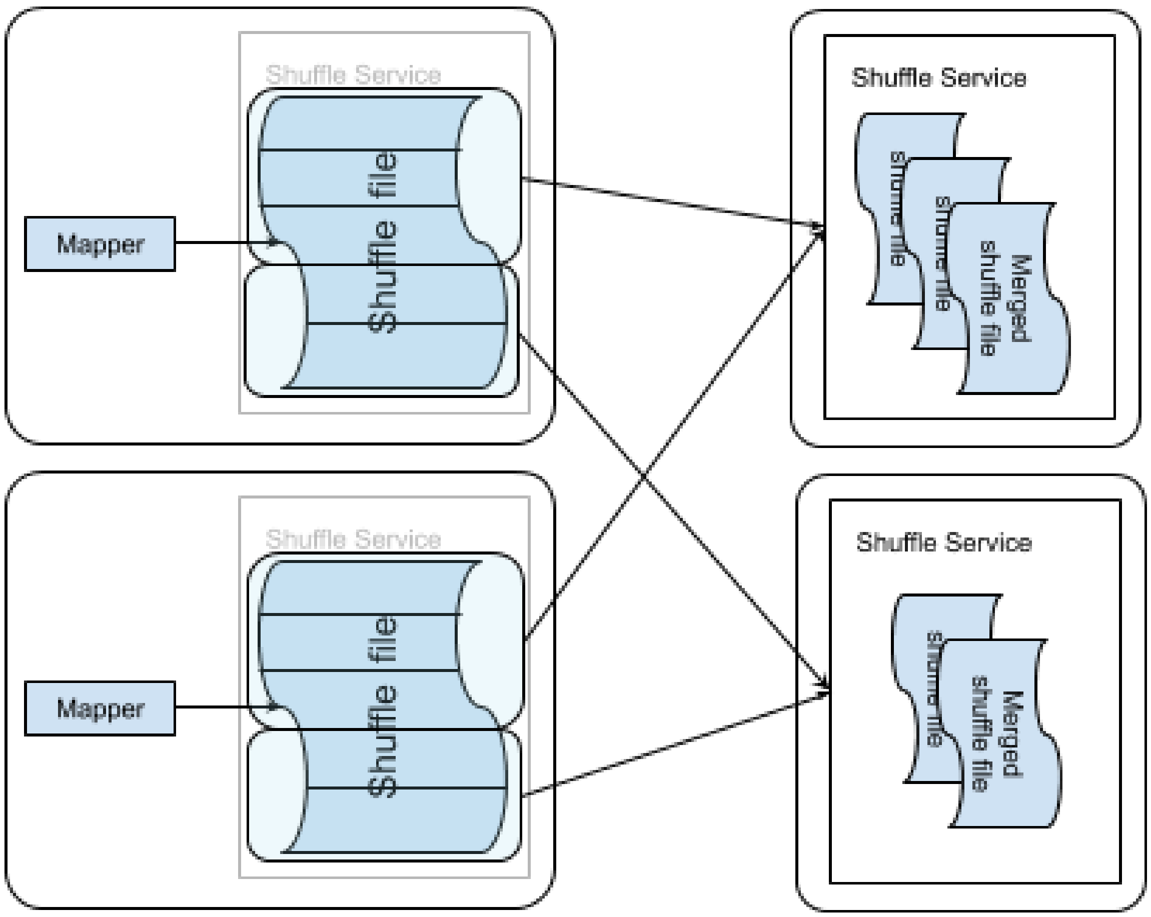


1. Spark DAGScheduler will determine a list of external shuffle services to use for a given shuffle before scheduling a ShuffleMapStage. This list of shuffle services could come from the current active list of executors or from the underlying cluster resource scheduler (more details discussed later). This list of shuffle services locations for a given shuffle will be saved on the driver side within ShuffleMapStage as well as sent to all the ShuffleMapTasks via ShuffleDependency. This guarantees that the driver and all the mapper tasks have a consistent view of the shuffle services to be used for a given shuffle.
2. The mapper task will proceed as usual till generating a pair of shuffle data file and index file. No change is needed here.
3. Before the mapper finishes and sends compressed MapStatus back to the driver, it would divide the shuffle partition blocks in the shuffle data file into multiple chunks, each containing multiple continuous shuffle blocks. The mapper or the ShuffleWriter has the partitionLengths array, with which it can tell the block boundaries in the shuffle data file. Since all mappers have the same list of remote shuffle services to talk to from ShuffleDependency, they can guarantee that the blocks for the same partition are sent to the same destination. When dividing blocks into larger chunks, the mapper would first divide blocks based on the # of remote shuffle services. It then groups blocks for the same destination into chunks based on their combined size. As mentioned earlier, we

can skip large blocks here in case of data skews. Once the blocks are divided into chunks, they are sent to the corresponding remote shuffle services to be merged into larger blocks per shuffle partition (more details in a later section). Even if the shuffle blocks are serialized and compressed, we can still merge them directly because the compression/serde encoders used by Spark support concatenation.

4. The mapper task only kicks off the block push process by sending the first chunk from its shuffle file. It would then finish and send the MapStatus back to the Spark driver the same way as the current mechanism. We rely on the Netty client threads to transfer the remaining chunks once acks are received on the client side. This way, the mapper task execution and the block push process are being handled by different thread pools. Because the task execution might be more CPU-heavy while the block push is more disk/network-heavy, this separation would bring benefits to further parallelize task execution and data transfer.
5. When the Spark DAGScheduler receives all mapper tasks' output at the end of the ShuffleMapStage, the block push process might not be fully completed yet. There is a batch of mapper tasks which just finished at this moment, and there could also be stragglers which are not finishing fast enough. Different from stragglers in the reducers, any delay we experience at the end of the ShuffleMapStage for all the blocks to be pushed/merged will directly impact the job's runtime. We thus expose configuration parameters to upper bound how much time the DAGScheduler waits before it starts scheduling the reducer stage. The wait is of course outside of DAGScheduler's event loop to make sure it's not blocking event handling. At the end of the wait, the DAGScheduler would notify all the previously selected shuffle services for the given shuffle (stored inside ShuffleMapStage) to finalize shuffle block merge and stop receiving new shuffle blocks. The shuffle services will also respond with a list of MergeStatus for all local merged partitions back to Spark driver. After collecting the MergeStatus from all shuffle services, the Spark driver knows about the location and metadata of every merged shuffle block in this shuffle. The Spark driver also already knows the location and size of every unmerged shuffle block from mappers' MapStatus. This way, the driver builds a complete picture about the shuffle blocks.
6. After the Spark driver gathers all MergeStatus, it posts a new event into the event loop so the DAGScheduler can pick it up and launch the reducer stage. We use the merged partition block's location as the preferred location for launching the reducers, since we can achieve much higher data locality now that the blocks are pre-merged. Each reducer would query the MapOutputTracker master endpoint for location and metadata for both merged and unmerged blocks it needs to fetch.
7. Once the reducer gets the information it needs, it starts sending requests to the corresponding shuffle service to fetch the shuffle blocks or reads the local shuffle blocks. We further allow the reducers to fall back to fetching the original unmerged blocks if it fails to fetch/read a merged block.

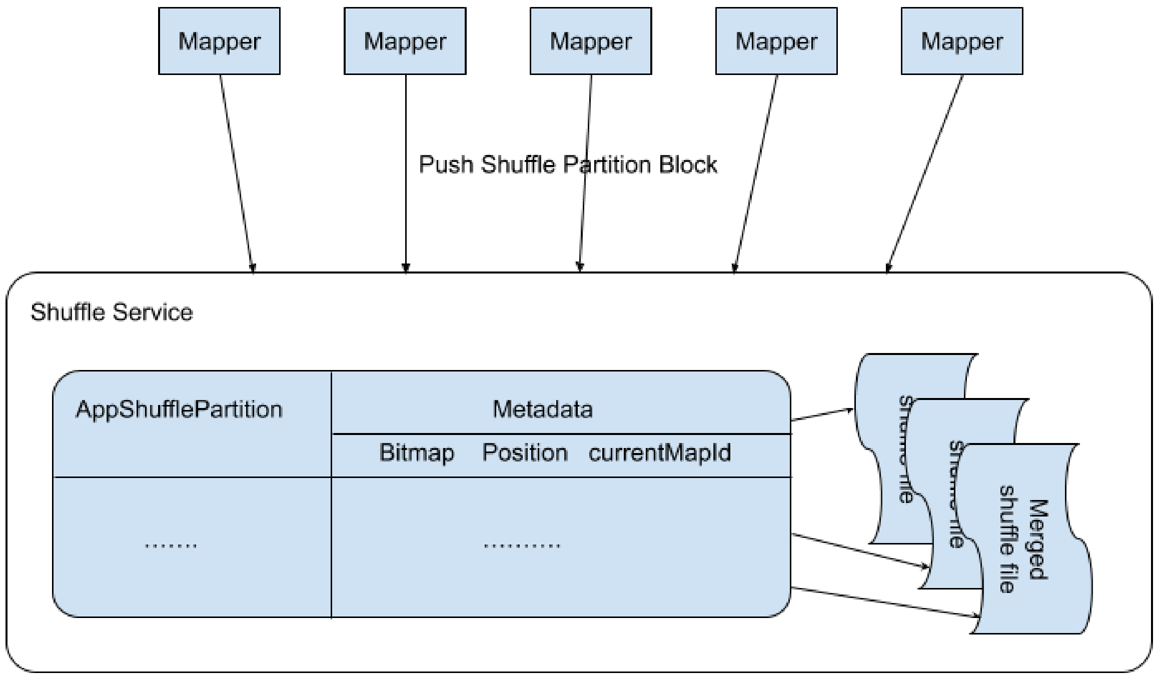
We further go over the design details of a few important components in the above mentioned steps. The first is the block push process in the shuffle write path.



Push-based shuffle: shuffle write path

The shuffle write path is depicted in the above diagram. The mapper will group blocks into chunks and send them to the corresponding remote destinations. In order to make this process efficient, each chunk needs to be read from disk in its entirety. This way, instead of reading blocks of 10s of KBs, chunks of a few MBs are read instead. In our current implementation, before transferring the chunk, we first cache it in memory on the executor side. Then individual blocks inside the chunk are sent to the remote shuffle services to be merged. This way, as soon as a block is handled by the remote shuffle service and ack'ed, the client can send more blocks to keep the netty channels busy.

An alternative way to implement this part is to read an entire chunk and send the entire chunk to the remote shuffle service in a single RPC. Along with the chunk data, we send metadata containing the block boundaries inside the chunk, so the remote shuffle service can handle each individual block in the chunk appropriately. This eliminates the need to cache data on the executor side, and it also generates much less # RPCs. However, it might make less efficient usage of the network since we are sending and acknowledging data at the granularity of chunks instead of blocks.



Shuffle service metadata for a shuffle partition

The shuffle service receives remotely pushed block data from multiple mappers for multiple shuffles of multiple applications. For each unique shuffle partition of a Spark application, it generates a merged shuffle file to append all the corresponding mapper-pushed blocks. It also maintains metadata for every shuffle partition that is currently being merged. The metadata contains a bitmap that tracks the mapper IDs of the partition blocks that are already merged, a position offset that tracks the offset after the most recently successfully appended shuffle block in the corresponding merged shuffle file, and a currentMapId that tracks the mapper ID of the current mapper whose partition block is being appended to the merged shuffle file. This metadata is keyed by the AppShufflePartition ID (application ID + shuffle ID + partition ID) and maintained as a ConcurrentHashMap. When the shuffle service receives a pushed block from any mapper, it would first retrieve the corresponding shuffle partition metadata. It then checks against the bitmap to make sure the block is not a duplicate of an already merged block. Then it would check if the block is coming from a mapper that matches the currentMapperId if currentMapperId is already set. Only the block coming from the currentMapperId can be appended to the merged shuffle file. This is because we are treating the pushed block as a stream, and a shuffle block might be sliced into multiple TCP packets. We need to guarantee that we are appending 1 shuffle block continuously into the merged shuffle file before writing the next block. Note that each mapper would randomize the order to transfer the chunks, thus the chance of multiple mappers sending blocks belonging to the same shuffle partition to the same shuffle service is very low. Once a block is successfully appended to the merged shuffle file, the position offset will be updated. Note that we could potentially write part of a shuffle block into the merged shuffle file before encountering failure on this shuffle block. Instead of corrupting the entire merged shuffle file, the position offset can help to ensure that the next block appended to the merged shuffle file can overwrite the corrupted portion, bringing the merged shuffle file back to a healthy state. If the corrupted block is the last block written to the merged shuffle file, the shuffle service can still truncate the corrupted portion when it receives requests from Spark driver to finalize the merge. Once a block is handled, either successfully or not, the shuffle service will respond back to the client about the result.

Another part to look further into is the shuffle read path. Because the reducers are scheduled with locality-awareness, many merged shuffle blocks are read locally directly. We still need to deal with fetching merged shuffle blocks remotely though since we cannot always achieve 100% reducer input locality. One consideration is whether to fetch an entire merged block in one RPC. Since a merged block represents most if not all the data for a shuffle partition, its size could be large. Fetching it all together could lead to some issues. While SPARK-19659 helps with the memory issue with fetching large blocks, fetching an entire merged block all at once would also make the parallelization of reducer task execution and block fetch worse. Right now, the Spark executor would use different threads to fetch remote blocks and execute reducer tasks on the fetched blocks. This way, while the remote blocks are being fetched, task executions already start. To address this issue, we can treat a merged shuffle block as a logical block consisting of many MB-sized sub-blocks. When the shuffle service is concatenating the mapper-pushed blocks into the merged shuffle block, it would carve out a new sub-block every few MBs and record these sub-block boundaries in an index file. Note that these sub-blocks are the results of concatenating multiple original blocks. When the reducer (ShuffleBlockFetcherIterator) fetches a merged block, we can introduce a new RPC so that the shuffle services can respond to the client the number of sub-blocks to fetch. This way, the reducer does not have to fetch the entirety of a merged block before it can start task execution. Note that, the metadata about how a merged block maps to a set of sub-blocks is managed by the shuffle service and not by the driver. This way, the driver only needs to track the logical blocks to save its memory footprint for tracking the merged blocks.

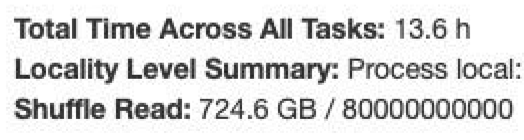
One last part to look into is how to determine the list of shuffle services for a given shuffle. For an on-prem cluster deployment, where the compute and storage nodes are the same, we might want to leverage data locality for fetching shuffle data as much as possible. To achieve this, it is important to make sure we are selecting shuffle services that are collocated with Spark executors. If dynamic allocation is not turned on, then DAGScheduler should just select the list of shuffle services collocated with the existing Spark executors. However, if dynamic allocation is turned on, at the beginning of a ShuffleMapStage, the # Spark executors might be less than the desired number. In this case, in addition to selecting the shuffle services collocated with the active executors, we can also leverage the underlying cluster scheduler to provide additional locations for shuffle services. Later on, when dynamic allocation kicks in, the dynamic allocation manager can request for resources based on the location of the pre-selected shuffle services. This way, we are effectively launching executors on the nodes where shuffle data merge is being performed. This increases the likelihood of shuffle data pushed to locations where reducers are running. When there are multiple concurrent shuffles going on at the same time, instead of selecting shuffle service locations for each shuffle independently, the DAGScheduler can potentially just reuse the locations of an active shuffle.

We also want to highlight a few benefits brought by this design.

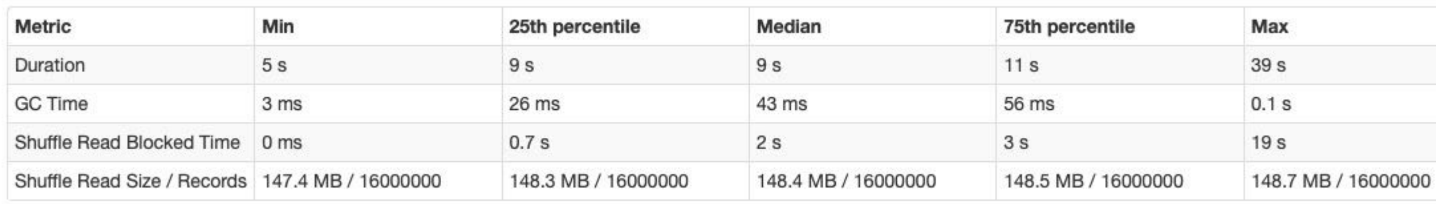
1. This approach helps to make the shuffle process much more efficient when served with HDD. It converts the small random reads to large sequential reads.
2. We designed the push-based shuffle as a complement to sort-based shuffle instead of fully replacing it. This means we are able to tolerate failures during the block push process, and are able to complete the shuffle even if not all the blocks are pushed. We are effectively generating a second replica of the shuffle data as part of the push-based shuffle process.
3. We are parallelizing task execution and shuffle data transfer on both the shuffle write and read path. On the write path, we are leveraging the netty client threads to handle the chunk push. On the read path, we are dividing a large merged block into multiple smaller yet MB-sized sub-blocks.
4. We let Spark DAGScheduler take full control of the block push process so it can cope well with stragglers. By early terminating the block push process, we can gain the efficiency improvements from push-based shuffle without suffering from the potential delays caused by stragglers. Once again, the 2-replica of the shuffle data allows partial push/merge of shuffle blocks.
5. Different from a few existing approaches, where the shuffle service might need to cache data in memory or perform sorting of records, we do not incur this memory/CPU overhead on the shuffle service side. The sorting is still performed by the mappers and reducers and we are only caching block data on the mapper side. In a busy cluster, a single shuffle service could receive thousands of connections from remote clients. If we let the shuffle service do sorting or cache data in memory, we might introduce a scaling bottleneck. Since most of the YARN cluster's CPU/memory resources should be given to executor containers instead of to YARN NMs, distributing such CPU/memory overhead into the executor side allows better scalability of the shuffle services.

With our prototype implementation, we have seen significant efficiency gain with push-based shuffle. In our benchmark, where ~725 GB of compressed data is shuffled between 8000 mappers and 5000 reducers, we have seen significant reduction of task runtimes as well as shuffle data that needs to be fetched remotely.

Before:

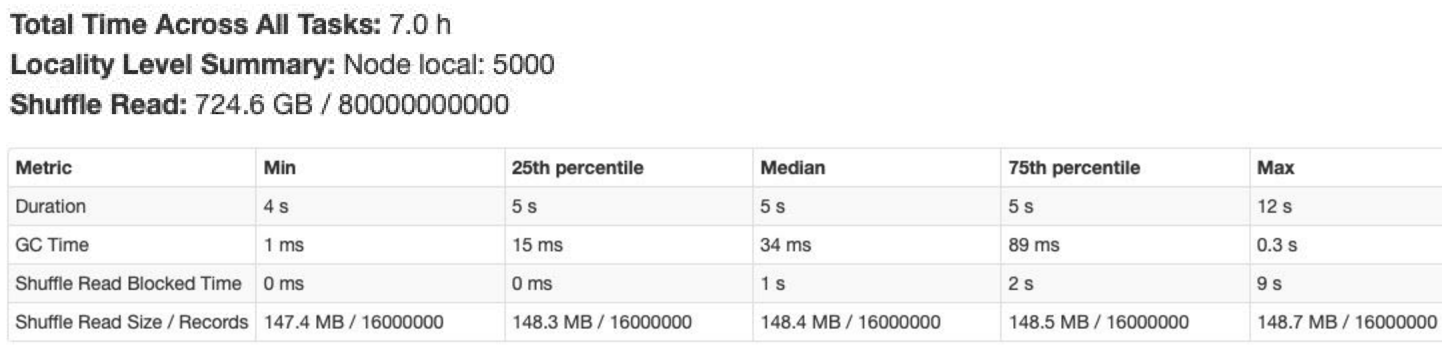


Total Time Across All Tasks: 13.6 h
Locality Level Summary: Process local: 5000
Shuffle Read: 724.6 GB / 80000000000



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	5 s	9 s	9 s	11 s	39 s
GC Time	3 ms	26 ms	43 ms	56 ms	0.1 s
Shuffle Read Blocked Time	0 ms	0.7 s	2 s	3 s	19 s
Shuffle Read Size / Records	147.4 MB / 16000000	148.3 MB / 16000000	148.4 MB / 16000000	148.5 MB / 16000000	148.7 MB / 16000000

After:



Total Time Across All Tasks: 7.0 h
Locality Level Summary: Node local: 5000
Shuffle Read: 724.6 GB / 80000000000

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	4 s	5 s	5 s	5 s	12 s
GC Time	1 ms	15 ms	34 ms	89 ms	0.3 s
Shuffle Read Blocked Time	0 ms	0 ms	1 s	2 s	9 s
Shuffle Read Size / Records	147.4 MB / 16000000	148.3 MB / 16000000	148.4 MB / 16000000	148.5 MB / 16000000	148.7 MB / 16000000

Appendix C. Optional Rejected Designs: What alternatives were considered? Why were they rejected? If no alternatives have been considered, the problem needs more thought.

For the high-level design, we could instead go with the approach to delegate shuffle storage to external storage layer leveraging more optimized storage mediums such as SSDs. We didn't take that route since it's not always practical to delegate a large-scale cluster's shuffle storage all to such storage mediums.

For the internal components of push-based shuffle, there are also a few design trade offs. For the shuffle write path, as discussed previously, we could take either an approach where we are reading an entire chunk and sending individual blocks or one where we read an entire chunk and send an entire chunk. We currently take the former approach for network utilization concerns. However, we are also interested in seeing whether we can further optimize here.

For the shuffle read path, we could also take a different approach to divide a large merged block into multiple smaller ones. One can imagine that multiple data files are created when merging a shuffle partition. Each is tracked as a separate block by the driver, so the reducers can fetch the merged shuffle partition the same way it fetches the non-merged blocks. While this approach helps to ensure we can reuse the existing RPC protocol to fetch merged blocks, it would create more overhead on the driver side to track multiple blocks for a merged shuffle partition.