Improving Spark Shuffle Reliability

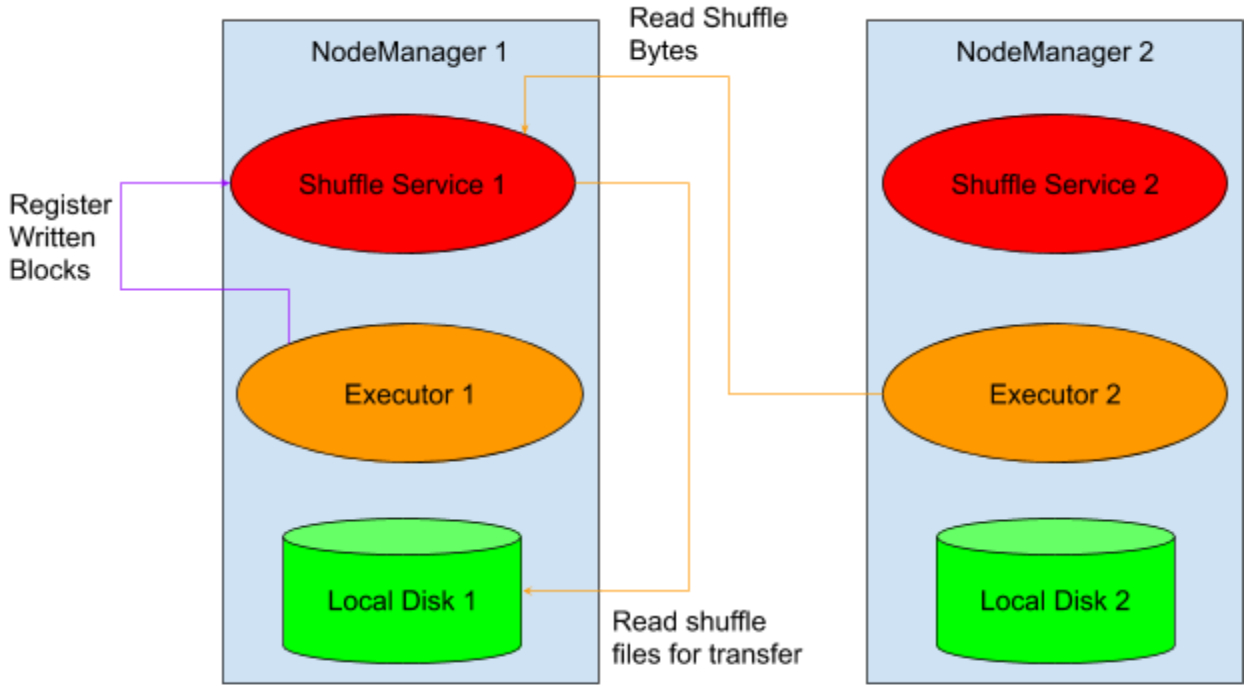
Background and Motivation

In distributed computing, the term *shuffle* is used to refer to the exchange of data between multiple computational units. Spark uses this term when data has to be reorganized amongst its executors, most commonly due to the need to repartition an RDD. Joining and aggregating RDDs by new partition keys is a common operation that requires shuffling the data; the items in the data set with the same partition key need to be colocated on the same executor node.

Much work has been invested in the performance and reliability improvements in Spark's shuffle operation. One important behavior of Spark's shuffle primitive is that Spark will write and read data to and from disk, and Spark applications need to maintain a record of the locations of these files on the different executor nodes. In Spark 1.2, contributors to Spark implemented an external shuffle service. The external shuffle service stores metadata about where shuffle files are located, as well as mappings from block ids to specific offsets within the shuffle files where the partitions are located. The shuffle service allows executors to continue transferring shuffle files even if some executors are terminated for any reason. Without the external shuffle service, the loss of executors would require recomputing blocks of the RDDs from scratch.

Problems With The Current External Shuffle Service

The current implementation of the external shuffle service uses the remote shuffle servers running on different nodes for reading data, while the executor processes still write shuffle data to their local storage. This architecture suffers from some shortcomings, particularly when it comes to its reliability and its usage in the context of containerized environments like Kubernetes and in applications running in Docker containers in YARN and Mesos. Below is a diagram of the shuffle operation being done with the current external shuffle service:



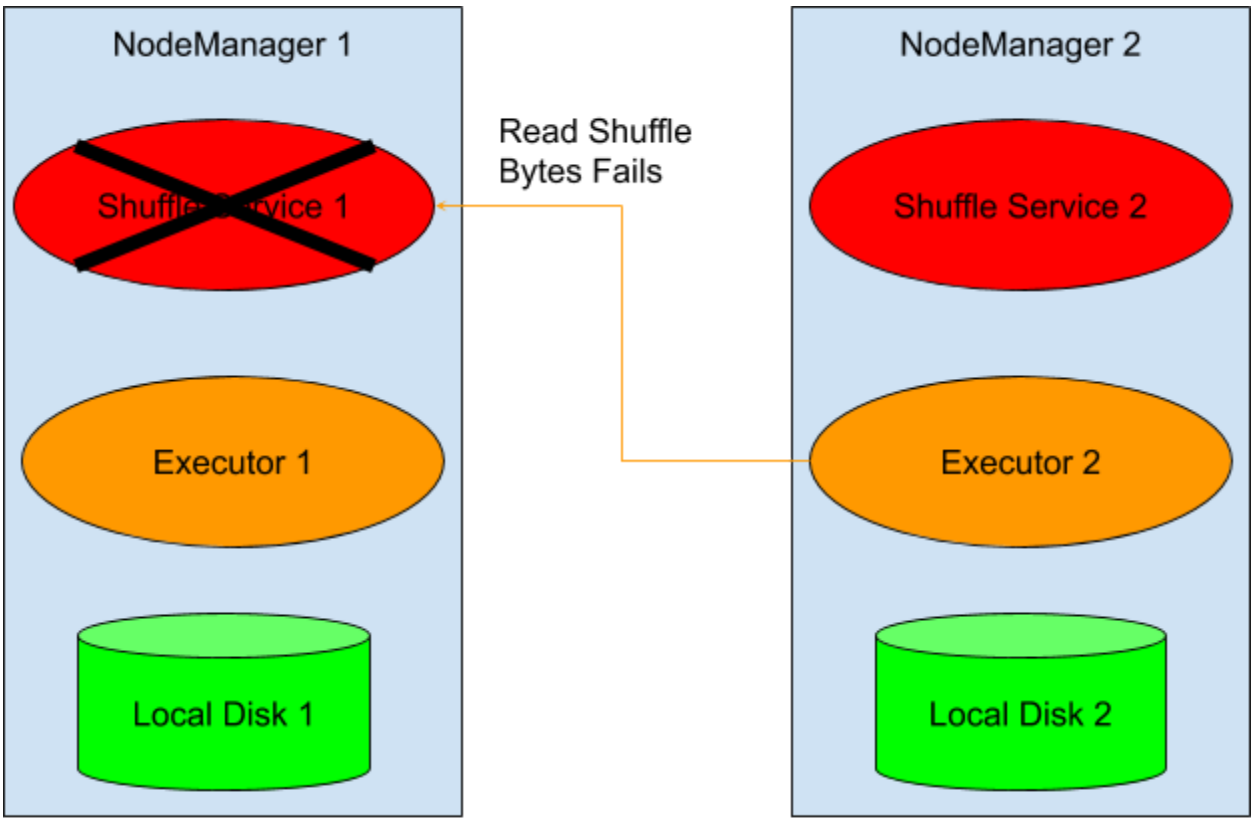
Shuffle data transfer between two executors in YARN. Note that if we remove executor 1, executor 2 can still read from shuffle service 1, and shuffle service 1 can still read from local disk 1.

The following problems exist with the current implementation:

1. **Lack of isolation**
    - The shuffle service runs on the same host as other processes that use the YARN NodeManager, including other Spark applications. If the shuffle service misbehaves, other applications using the YARN cluster may be negatively impacted.
    - The shuffle service runs inside the NodeManager process itself, so if the shuffle service runs out of memory, the NodeManager becomes unavailable and the cluster will have fewer workers for resources.
2. **Scalability issues**

- All executors from various Spark applications will write to the same local disk. Conversely, a single shuffle service will have to serve numerous read requests from executors across all of the running Spark applications. It is difficult to balance the load of disk pressure across the cluster, so one node can conceivably be overwhelmed with disk pressure if that node runs many executors.

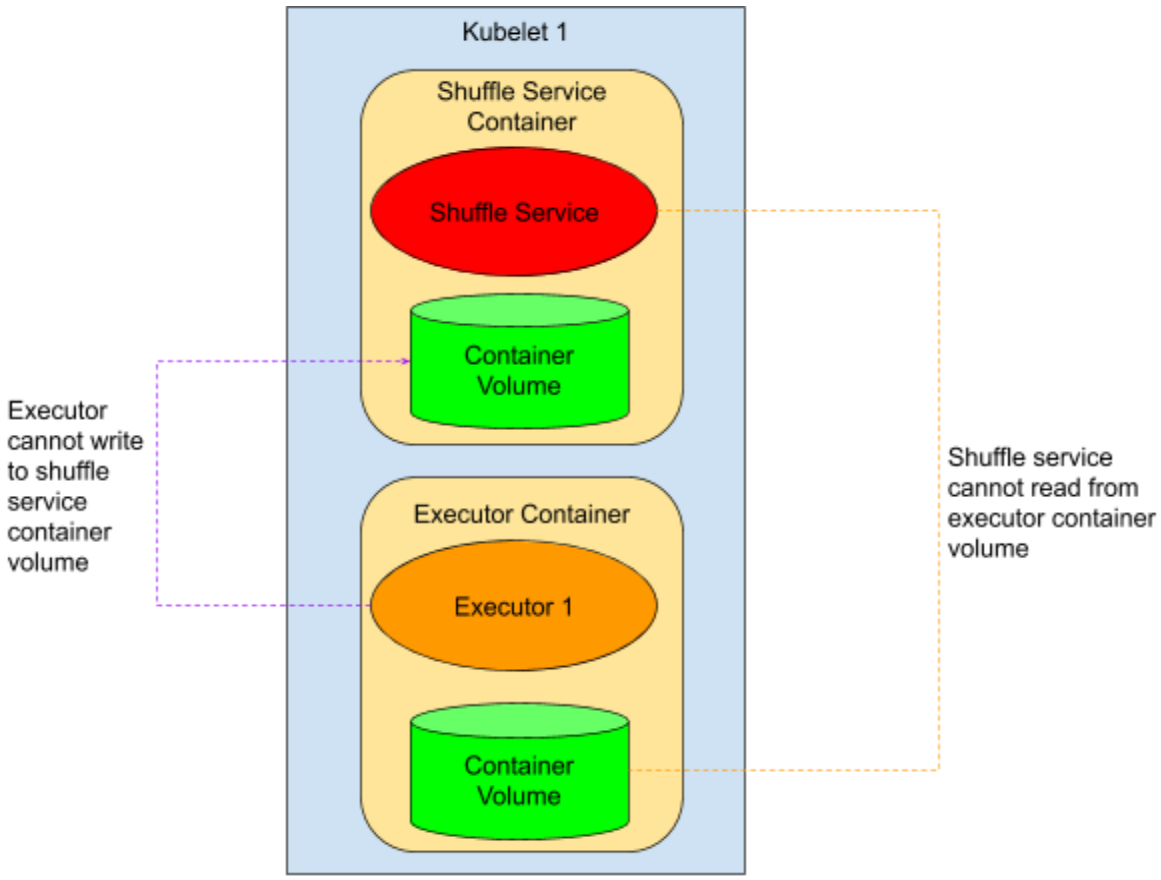
3. **Lack of replication leads to recomputing lineage on node failure**
   - An executor will only write data to its local storage, and the data is not replicated across multiple nodes. This means that if the shuffle service that was colocated with that executor crashes, the shuffle data written by all executors on that node is lost. Every Spark application that wrote shuffle files on that node will need to recompute the lost work.



Notice if shuffle service 1 goes down, the shuffle read will fail. The application has to recompute the shuffle files that were stored in NodeManager 1.

4. **Assumptions of colocated storage do not always hold in containerized environments**
   - When the executor writes its shuffle data, the executor assumes that the files it writes to will be available for its colocated shuffle service process to be able to read. However, in Kubernetes and in Spark applications running in Docker

containers on YARN and Mesos, this assumption does not necessarily hold. Kubernetes and YARN both support mounting local volumes into containers. Unfortunately, not all clusters can support this feature. A cluster administrator may enforce a policy on a cluster that all containers must be fully isolated from one another, in order to maximize isolation between the components. This can be the security posture taken by cluster administrators who expect untrusted user code to run in the containers.



The isolation between the shuffle service container and the executor container renders the current implementation unusable in Kubernetes. Sharing volume mounts between containers should be avoided.

5. **Requires continuous uptime**
    - Every executor requires a colocated external shuffle service to be running. If an external shuffle service is stopped, the node that shuffle service is running on cannot be used to schedule executors.
    - Shuffle services that are running but not serving executors are wasting compute resources in the cluster.

Design Objectives

In this document, we aim to discuss potential changes to the way Spark shuffle works to alleviate these problems. The problems described above can be converted into objectives that we aim to achieve with our modifications:

1. It should be possible (though not strictly necessary on every installation) to isolate every component in the system such that if one component fails, at most one Spark application can fail..
    a. Note that currently, Spark drivers and executors can theoretically be completely isolated by running them in containers on Kubernetes or dockerized Mesos / YARN. Thus the only component that is lacking in the isolation department is the external shuffle service.
2. Shuffles should scale regardless of the number of applications and the number of executors that are running.
    a. Shuffle scales well for a single application already - this is one of the reasons why Spark is a high-performance, industry-grade distributed computing framework. However, in edge cases where many applications and executors are sharing the same shuffle service, we encounter issues as described previously.
3. Failures or shutting down any one host in the system should not require any Spark applications to recompute RDD blocks.
    a. This covers problems 3 and 5 above.
4. Resilient shuffle should be possible even in containerized environments where all processes are completely isolated from one another, save for specific API endpoints (e.g. over RPC).
5. Load should not be able to overwhelm a single component. There must be no bottlenecks.

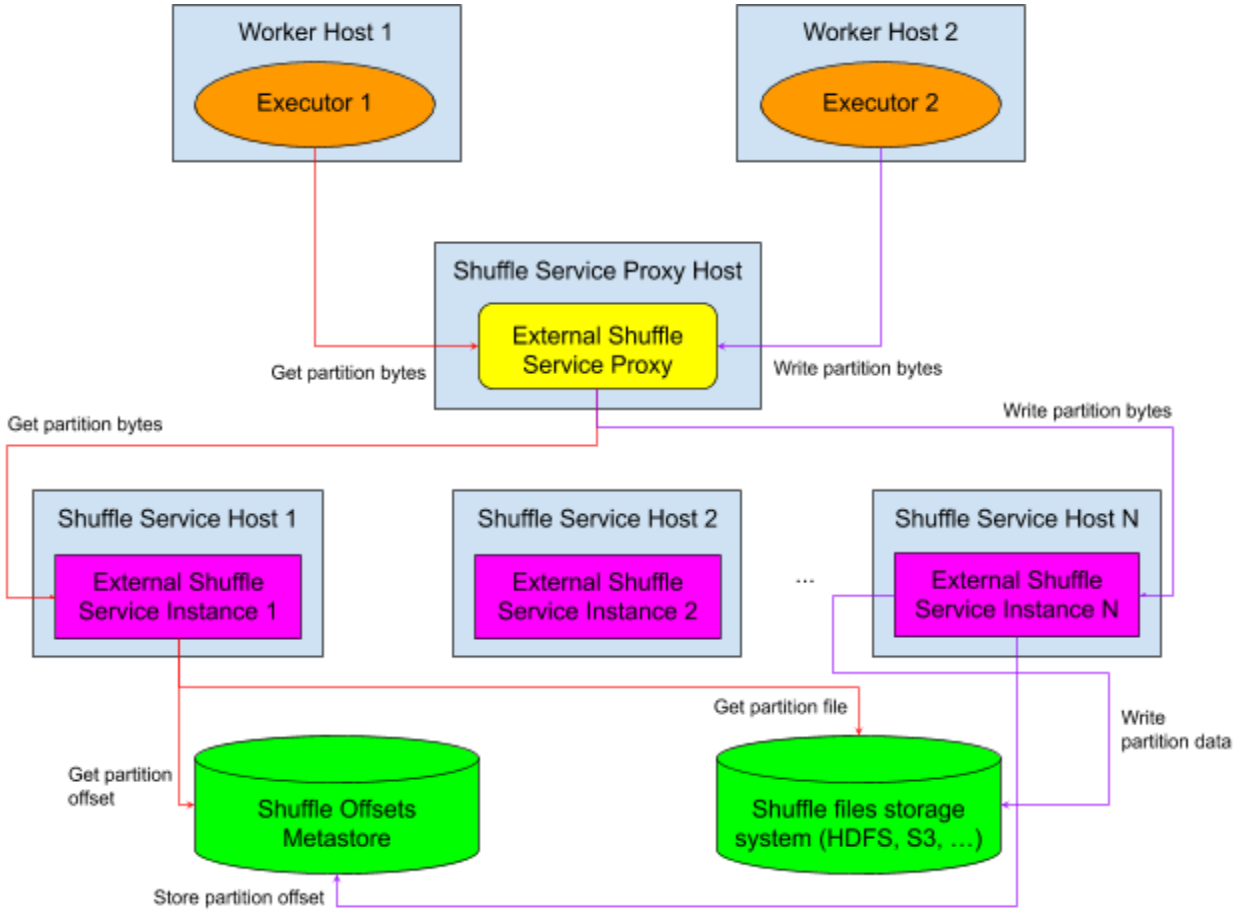
General Premise And Proposed Solutions

To solve all of the above problems, we propose that executors should not be constrained to store shuffle data to the executor's local disk. There are three architectural alternatives that may be considered to relax the existing constraints.

Option 1: Upload Shuffle Data Through The Shuffle Service

Treat the shuffle service as a file server that accepts streams of bytes as input for writing shuffle data, and provides endpoints for reading these shuffle files as byte streams. The shuffle service can be thought of as being similar to a highly available web service. The architecture will operate as follows:

1. The shuffle service is started on some set of nodes.
2. The shuffle service is configured with some remote file system to put shuffle files into.
3. All shuffle service instances share some database / metastore that tracks the offsets within shuffle files corresponding to RDD blocks. The path to a shuffle file can be derived as it is currently (combination of executor id, shuffle id, map id).
4. Spark applications are given a URI for accessing an external shuffle service.
    a. This can be a stable URI to a load balancer / proxy that routes to some external shuffle service instance started in #1.
    b. All shuffle service nodes will behave equivalently, so the proxy can pick any of them.
5. When executors write shuffle files, they call an endpoint from the URI given in step 4 to open a writable byte stream.
6. When executors read shuffle files, they call an endpoint from the URI given in step 4 to open a readable byte stream.



In considering if this solution achieves the above objectives:

1. Every external shuffle service can be run in its own host or in an isolated container. If any external shuffle service instance fails, only the container running the shuffle service will be affected. It is not necessary in this implementation to colocate shuffle services

with Spark executors. This solution therefore solves the problems described around isolation.
2. As the number of executors grows in the system, it is possible to independently scale the metastore, the shuffle files storage system, and the shuffle service to accommodate the busier cluster. The shuffle service is placed behind a load balancer that ensures no one shuffle service is overloaded with work. This solution therefore solves the problems described around scalability.
3. As before, executors can shut down freely because their computed shuffle blocks are persisted in a separate storage system. In this case, however, the shuffle service instances can shut down as well, and the underlying data can be serviced by any remaining external shuffle service instances. This solution therefore solves the problems described around reliability, downtime, and wasted work.
4. The external shuffle service does not need to share local storage with the executors; in fact, the shuffle services themselves are completely stateless. Therefore as long as the metastore and the shuffle files storage system can be run in containers (for example, using the cloud as the backing store - though there are other options), this solution is compatible with containerized and fully-isolated runtimes.
5. All executors can call into different shuffle service instances, with the load balancer ensuring roughly equal work being distributed to the different shuffle service instances. There does not appear to be a single bottleneck in the system.

Implementation details to consider:

1. Security: Spark applications shouldn't be able to call endpoints to read each other's shuffle data. Spark applications shouldn't be able to pollute shuffle data from other applications. Transferred bytes should be encrypted - TLS? SASL?
2. Endpoints protocol - HTTP? Netty?
3. How to implement the backing store for the shuffle files?
    a. Store shuffle data in HDFS? S3?
    b. Shuffle service nodes gossip amongst each other to replicate shuffle data?
4. How to implement the shuffle file locations / index storage?
    a. Metadata / Index files in some highly available storage system? HDFS? S3?
    b. PostGres / MySQL / Cassandra? What is the schema?
5. File cleanup
    a. When should the shuffle service purge shuffle data? The Spark driver can call some endpoint upon shutdown, but what if the cluster manager kill -9s the driver? Maybe heartbeat / polling?

Advantages:

- Backend implementation of shuffle file indexing / caching / backing storage is completely transparent to the Spark applications. Configuration on the Spark application side is minimal.

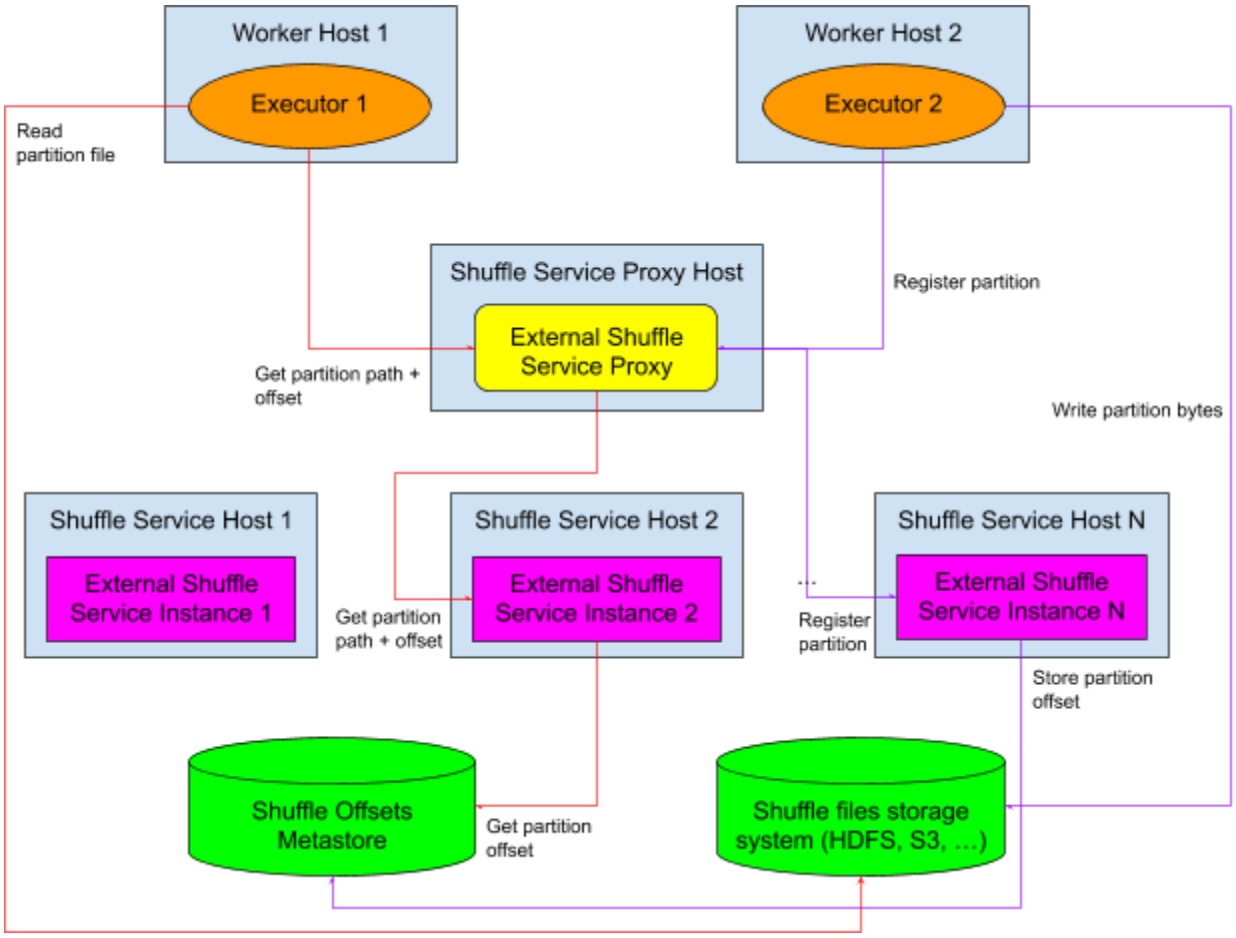
Disadvantages:

- Each shuffle file has to be written in three hops - once to the shuffle service proxy, once to the shuffle service backend, and once to the backing file store.

Option 2: Shuffle service Provides Remote File URIs

The shuffle service continues to maintain file paths, but executors treat these paths as being located on some remote file system instead of being local to the executor itself. The shuffle service uses a database to store these paths for each application. The architecture will work as follows:

1. The shuffle service is started on some set of nodes.
2. All shuffle service instances share some database / metastore that tracks the paths of shuffle files, as well as the offsets within shuffle files corresponding to RDD blocks.
3. Spark applications are given a URI for accessing an external shuffle service.
    a. This can be a stable URI to a load balancer / proxy that routes to some external shuffle service instance started in #1.
    b. All shuffle service nodes will behave equivalently; the proxy can pick any of them.
4. Spark applications are configured with a “root” URI for all shuffle data to be written to for that application. E.g:
   `hdfs://my-cluster.spark.test:9000/application-1249012851/tmp/`
5. Whenever Spark executors write shuffle data, the executor opens files with a root of the URI configured in step #4.
    a. The executor will write the data into the remote storage layer.
    b. The executor derives the path to write to based on current logic. The path is then resolved against the root URI with the external shuffle service.
6. Whenever executors read a shuffle block, the executor contacts the external shuffle service to find the partition’s byte offset for that block. The path to the shuffle file is derived from the executor, shuffle id, and map id, as the current implementation does. The executor then separately resolves the path against the root URI and reads the data directly from the remote file system.



In considering if this solution achieves the above objectives - we note that this solution has the exact same architecture as option #1, but the only difference is the APIs of the external shuffle service and the interactions between the different programs. The same line of reasoning that shows that option #1 satisfies all the required criteria also applies to this implementation.
Implementation details to consider:

1. Security: Spark applications shouldn't be able to call endpoints to read each other's shuffle data. Spark applications shouldn't be able to pollute shuffle data from other applications. Transferred bytes should be encrypted.
2. How to implement the shuffle offsets storage?
    a. Metadata / Index files in some highly available storage system? HDFS? S3?
    b. PostGres / MySQL / Cassandra? What is the schema?
3. File cleanup
    a. When should the shuffle service purge shuffle data? The Spark driver can call some endpoint upon shutdown, but what if the cluster manager kill -9s the driver? Maybe heartbeat / polling?

Advantages:

- Only metadata has to travel through multiple hops.

- Smaller change required compared to #1
    - Replacing all references to File to some remote file implementation should suffice. The shuffle service would just store the index files in some distributed file system instead of on local disk, and the shuffle block writer and reader use remote files instead of local files.
    - Incremental changes for faster access to partition metadata can be independently done in the shuffle service backend code, without changing the shuffle service's metadata location API.

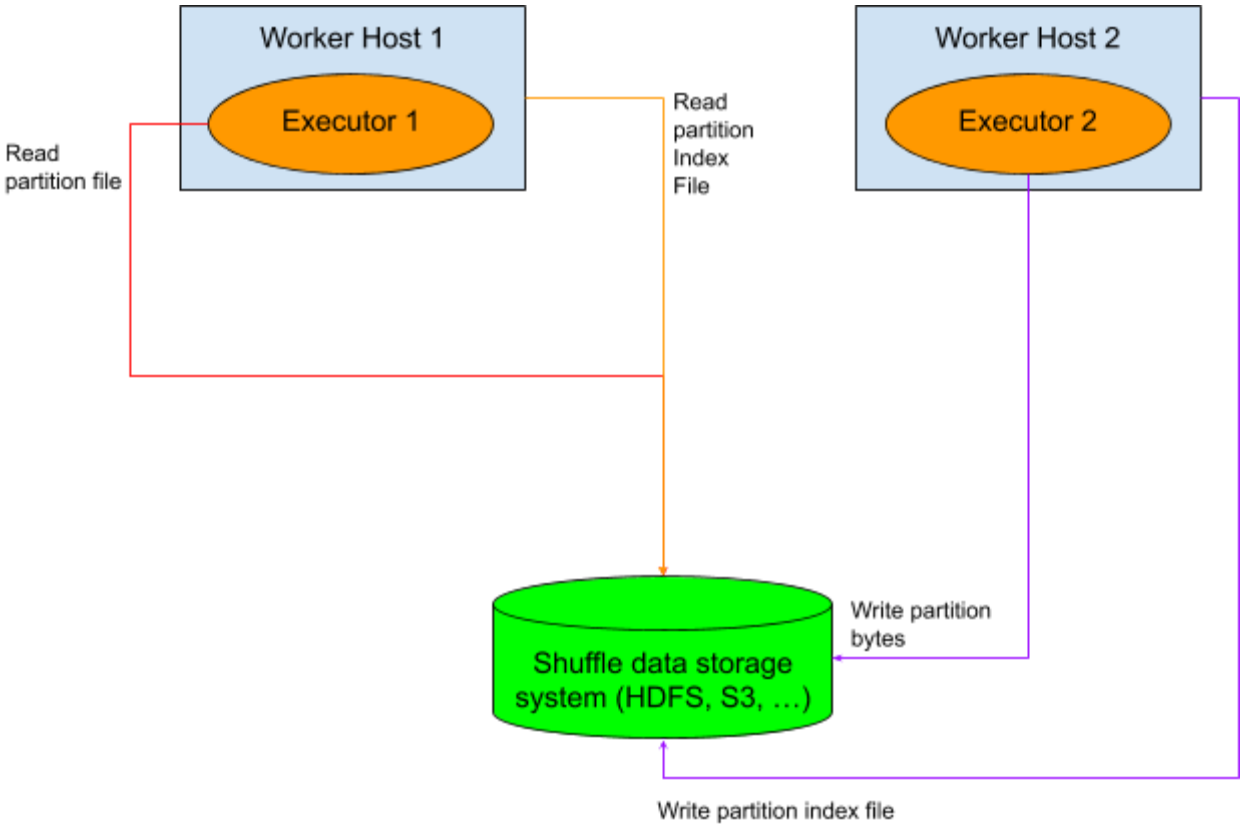
Disadvantages:

- Adds more complexity to Spark application configuration - need to select the backing store for shuffle files.
- Requires all applications to have direct access to the backing store. In option #1, access to the backing store for temporary shuffle data could be restricted via firewalls, etc. - the shuffle service is a single point of entry for accessing the shuffle data. In this approach, any application with access to the backing cluster can access the shuffle data.
    - Proper security measures applied to the backing file store itself should mitigate this significantly.

Option 3: Driver maintains all of its own shuffle metadata without a shuffle service

The application itself can maintain all of its own information about the application's shuffle files, thus removing the need for an external shuffle service at all. The implementation would roughly work as follows:

1. The application itself keeps track of the mapping between partition ids to file locations + offsets within the file.
2. All file locations are in some remote file storage layer.
3. Executors open and close streams for shuffle files directly against the backing storage system.



In considering if this solution achieves the above objectives:

1. In this solution we've removed the external shuffle service, so the only remaining components are the shuffle data storage system and the Spark applications themselves. Spark applications can be isolated, while the storage system can be separated from the computational jobs that access its data. This solution solves the problems described around isolation.
2. Since we don't have an external shuffle service, the scalability of this solution falls to the scalability of the shuffle operation of Spark in general as well as the scalability of the shuffle data storage system. The latter is a well known field with plenty of existing work, while the former may be improved but as a separate effort.
3. Executors can shut down freely, and the written shuffle data will still be available via the persistent backing store. We no longer have to worry about the uptime of a separate system. This solution therefore solves the problems described around reliability, downtime, and wasted work.
4. Spark applications running inside containers can access the remote storage system for shuffle read and write. This solution is compatible with containerized runtimes.
5. As long as shuffle blocks are replicated across nodes in the distributed file system, there should not be a single bottleneck in this system. We don't have a single component

loading all of the index files at once - all the executors derive the metadata and the data they need to fetch in a distributed manner.

Implementation details to consider:

1. How would the driver maintain all of the relevant data in a scalable way?
    a. Storing in driver only is probably too expensive - need to avoid making the driver a bottleneck.
    b. Use some external metastore that the executors can access - but what kind of metastore? Files in distributed storage? Database?
    c. The current solution actually still works here - the convention mapping locates the file location and the index file stores the offset, and the index file is also located with convention mapping.
2. File cleanup
    a. If the driver exits unexpectedly, how do we purge the written shuffle data from the backing storage? It's particularly difficult in this case without a shutdown hook since there is no third party that knows about the written shuffle files.
3. Security - encrypt the shuffle data, set remote filesystem permissions

Advantages:

- Greatly simplified infrastructure and architecture. No need to worry about a separate service.

Disadvantages:

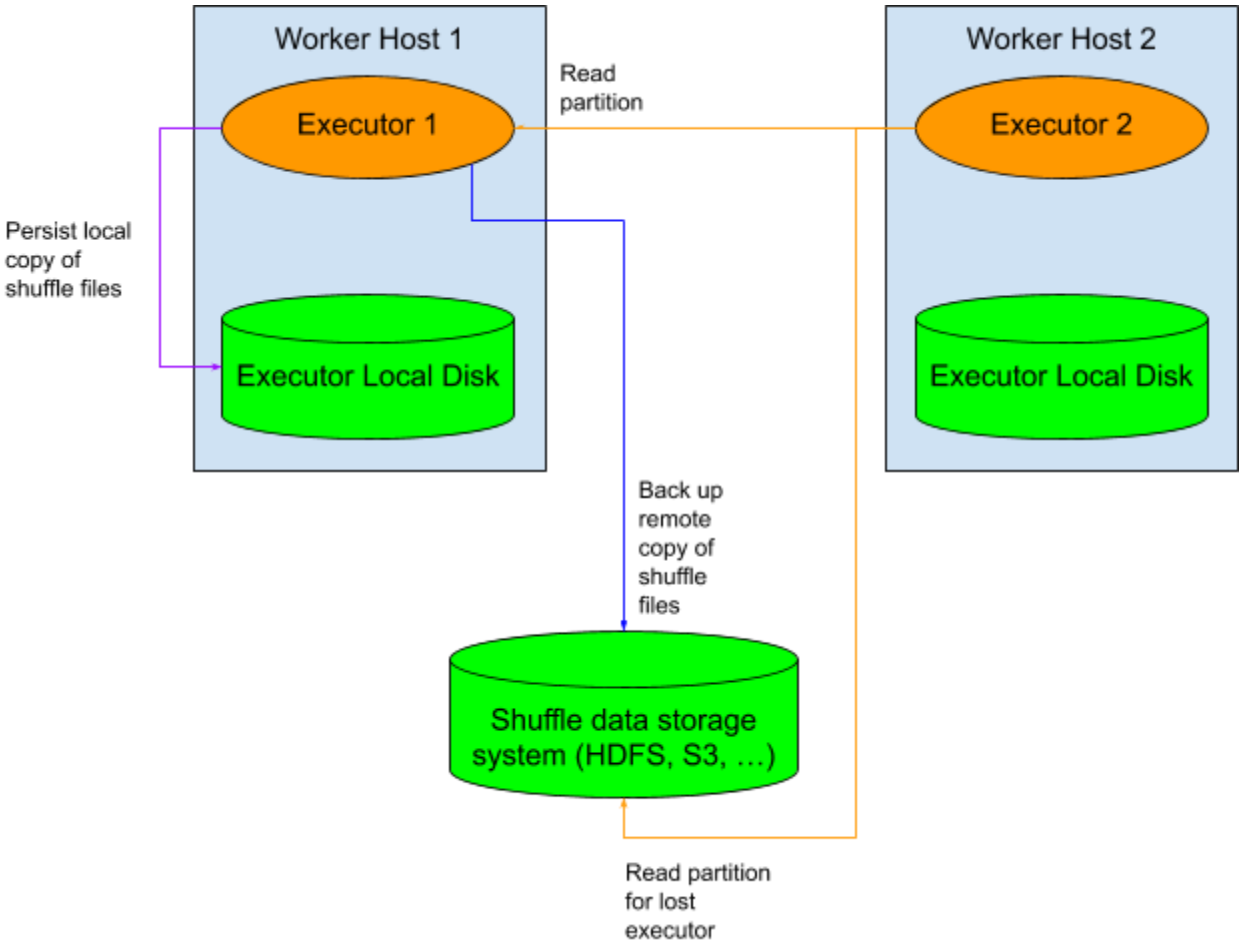
- Cleaning up data is unreliable. Suppose the Spark application receives a kill -9 signal. In such a case, the Spark application cannot clean up its own shuffle files. There is no other component that knows about the existence and lifecycle of such files. Therefore the files could potentially be left there indefinitely.
- Efficiently reading index files is harder. Previously, if two different executors asked the shuffle service for the same index file, the shuffle service could cache the index file between calls. However, now the shuffle metadata has to be read from the backing store twice - once for each executor. Cross-executor caching of the index files becomes harder.
    - Alleviated somewhat by distributed storage which supports caching natively.

Option 4: Back Up Shuffle Files to Distributed File System

Instead of requiring the application to write to the distributed file system directly, shuffle files can be written to the local disk of executors at first and then later asynchronously uploaded to a distributed file system. If the executor dies but its shuffle files have been written to the distributed store as a backup, the backed up files can still be accessed.

1. Executors write shuffle files to local disk. They also continually upload shuffle data and index files asynchronously to a distributed file system.

2. When executors need to exchange shuffle data, they contact one another directly to read the data (as if there was no shuffle service present in the current model).
3. If an executor is not reachable, the executor reading the shuffle data attempts to read the shuffle file and shuffle index file from HDFS, if available, using a canonical file location.



In considering if this solution achieves the desired objectives:

1. We've removed the external shuffle service here as well, so if one executor fails it will only impact the application that executor is a part of. The backup storage layer should be resilient and highly available.
2. One potential problem is if an executor receives a lot of shuffle data and has to spend a lot of I/O uploading all the backups asynchronously. This isn't problem until we consider the fact that if this executor crashes without having backed up all its shuffle files, that shuffle data must be recomputed. Therefore a large shuffle that required backing up a lot of data is more likely to lose some shuffle blocks permanently, so this solution leaves something to be desired on the scalability front.
3. As discussed above, if an executor fails before it has finished backing up its shuffle files, the shuffle blocks that have not been backed up need to be recomputed.

4. This solution is compatible with containerized storage, since the local disk does not need to be shared between any two processes in the system. Exchanging shuffle files is done over RPCs between executors or between the executor and the remote backup storage.
5. There is no single bottleneck in the system.

Implementation details to consider:

- Is there a version of this that can use an external shuffle service? Why would we want to do so?
    - For example, suppose we uploaded data through the external shuffle service like we do in option #1, but instead of the shuffle service writing to remote storage synchronously, the shuffle service cached a local copy of the file and then wrote to the backup storage asynchronously.
    - Main advantage is that shuffle service pods should be more resilient than Spark executors on average, given the heavier workload that executors are subjected to.
    - Requires executors to find the specific shuffle service instance to read from, or else a shuffle service proxy to track the locations + issue a redirect.
    - Requires network write to persist the shuffle data instead of local storage write.
    - However now if a shuffle service node goes down before it backed up all its data, multiple applications are impacted - namely, all applications that wrote shuffle data to that shuffle service.

Advantages:

- Fewer network hops in the best case scenario when executors do not crash - shuffle write can be to local disk instead of having to go over the network to distributed storage.

Disadvantages:

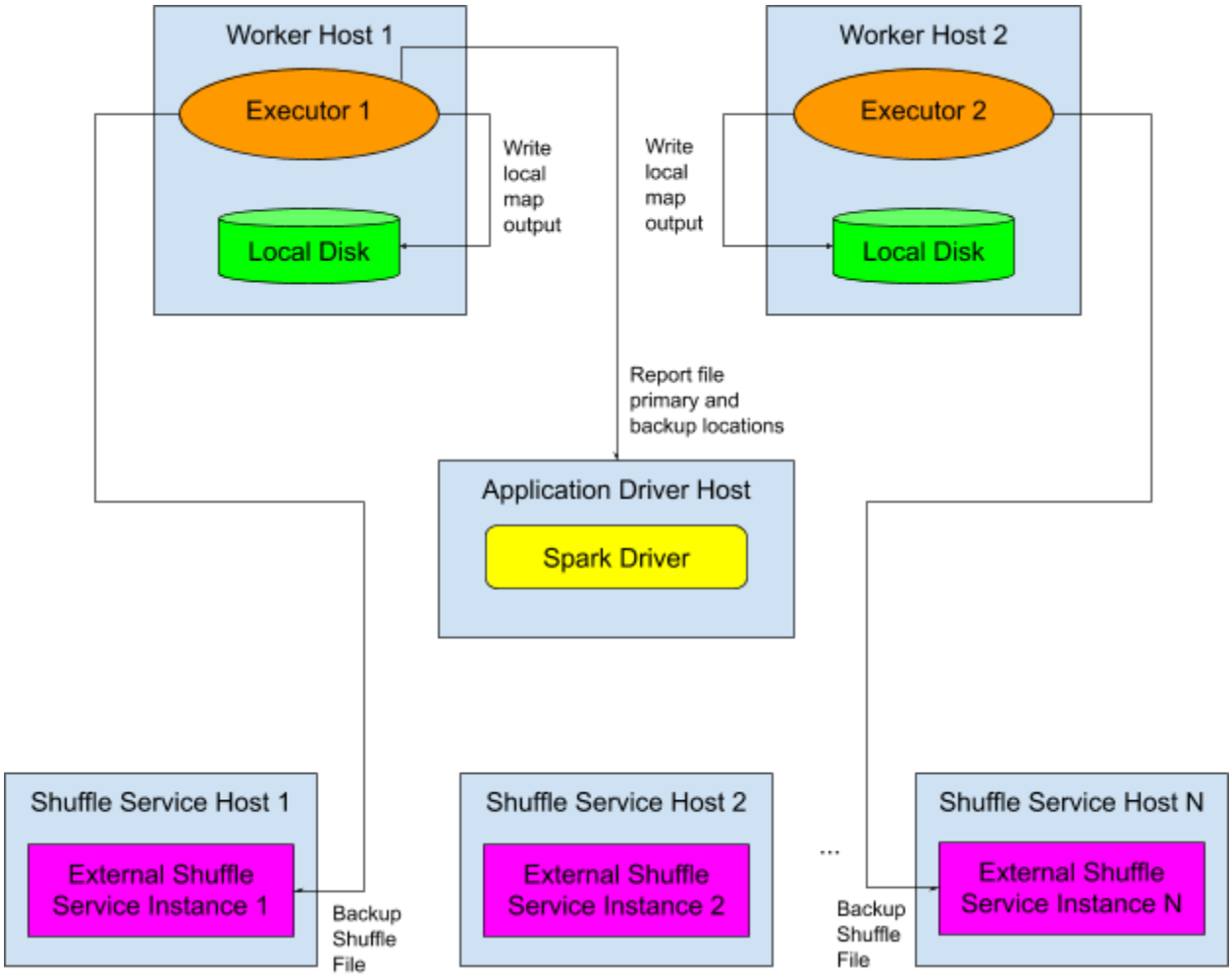
- Recomputing lineage is required when the executor crashes before it has finished backing up data.

Option 5: Upload Data To The External Shuffle Service And Have The Driver Track File Locations

The shuffle service can be responsible for storing the shuffle data, but the driver can be responsible for tracking the shuffle service hosts that the files have been replicated on. Designs 1-4 described above require usage of a distributed file system, but most distributed file systems require a master node to track the metadata about how the files are distributed across the cluster. This master node would have to scale with the number of files in the distributed file system, which could grow large if that cluster would need to store shuffle files from all Spark applications that are running in the system.

The following design proposes separating out the shuffle file tracking to the Spark applications themselves. The shuffle service just behaves like a simple file server, and the applications are responsible for using these file servers to provide redundancy for the shuffle data. The design would work as follows:

1. Executors write shuffle map output and index files to local disk and report map outputs to the driver as before.
2. Executors also upload the shuffle map output and index files to the external shuffle service as a backup(s).
    a. The driver has to locate instances of the external shuffle service in the cluster. This will be cluster manager specific; for example for YARN there should be a shuffle service running on every node manager host, but for Kubernetes one can specify a set of pods matching a label in a namespace, then connect to the shuffle service by pod IP..
    b. The driver or executors can pick the shuffle service hosts to use for backups.
    c. Backups can be asynchronous for slightly less resiliency to executor failures (e.g. if no backups complete before the executor crashes, the shuffle data is lost forever).
    d. Every shuffle file can be replicated to any number of shuffle service instances.
3. Executors inform drivers of the locations of the backed up shuffle files. The map output tracker holds metadata about these backups as well.
4. Reducers attempt to read shuffle data from the executors directly first. If that fails, the shuffle files are read from the backup.



In considering if the above solution achieves the above objectives:

1. There is some redundancy in this solution that makes it more reliable than the previous solution. Previously, when an external shuffle service crashes, the map outputs from all executors that shared data with that shuffle service would be lost. In this case, there is no strict requirement for the shuffle service that backs up the map outputs to be on the same host as the executor. Furthermore we show preference for the executor itself to serve its own shuffle data. Therefore when any given external shuffle service crashes, the executors that backed up data to that shuffle service could just continue serving its own shuffle data. The resiliency can be improved further by having executors back up shuffle files to multiple shuffle service instances. The extra redundancy gives us greater resiliency in the case where any one component fails in the system apart from the driver. Therefore this solution achieves objective #1.
2. There are two main ways we can evaluate the scalability of this solution, in terms of scaling with the number of applications in the cluster - that is,

    a. Backing up shuffle data leads to an overall increase in disk utilization across the cluster by a significant amount. One option to reduce the disk usage is for backups to be stored in highly compressed formats. The idea here is that the backups should only need to be referenced when the executors can no longer provide that shuffle data, so we don't always need to pay the cost of decompression.
    b. Other solutions that require a distributed file system with a single master node require that master node to hold metadata about all files in the file system. For example, the HDFS namenode does not scale well for large numbers of small files in the cluster. This is problematic because multiple Spark applications can write large numbers of small files, creating problems for the Hadoop cluster. The advantage of this solution is that no single component has to track all of the shuffle files across all Spark applications. The Spark driver is only responsible for tracking its own backup shuffle files. However, this puts greater strain on each individual Spark application, because the application needs to hold more in-memory state about the primary *and* backup map output locations.
3. Shuffle files can be replicated such that if one shuffle service is lost, other shuffle service instances can provide the lost blocks. This solution achieves objective #3.
4. The backups are written over the network, and no two components need to share the same disk space. This solution can run in a containerized environment, and thus achieves objective #4.
5. As mentioned earlier, it is unclear if the driver will be able to handle tracking both the primary and the backup map output metadata. We already have observed large numbers of partitions giving Spark drivers trouble. The memory footprint of drivers may increase drastically; we can explore on-disk storage options for map output metadata.

Implementation details to consider:

- Synchronous or asynchronous backup?
    - If synchronous, what will be the impact on job runtime?
    - If asynchronous, how do we handle the executor exiting before it has finished all its backups?
- Storage / file format? Can we compress to reduce disk usage?
- Encryption / Authorization
- File cleanup - must clean up the shuffle files when the application finishes / crashes.
- Does the executor itself even need to serve the shuffle data in the first place? What if the executor just wrote the shuffle file directly to the shuffle service?

Advantages:

- No dependency on a third party distributed storage solution like HDFS, meaning we have full control of read/write performance.
- No single node responsible for storing all shuffle file metadata.

Disadvantages:

- Disk usage blowup when the files are replicated, particularly when both the executor and the shuffle service are each storing a copy of the data.
- As mentioned before, the driver will have to store both primary and backup map output information, putting more strain on the driver.

A Word About Performance

One crucial difference between the original shuffle logic and designs 1-4 is that shuffle data is now going to be read and written over the network. Previously, shuffle data could be written directly to the executor's local disk, while that data would be read over the network.
After we build any of these solutions, it is critical to verify the performance impact caused by the extra network interactions. In principle, the network stack has become much faster over the past few years. We also note that there has been previous work done[1] in evaluating the performance impact of switching shuffle writes to a network write operation. Some additional optimizations may need to be introduced to the shuffle operation itself. There has been some previous work and research done in that domain as well.

References

1. Taking Advantage of a Disaggregated Storage and Compute Architecture https://databricks.com/session/taking-advantage-of-a-disaggregated-storage-and-compute-architecture
2. SOS: Optimizing Shuffle I/O https://databricks.com/session/sos-optimizing-shuffle-i-o