SPIP: SPARK-25299 - An API For Writing Shuffle Data To Remote Storage

Authors: Matt Cheah and Yifei Huang

Acknowledgements: Special thanks to Ilan Filonenko for writing the initial individual file server prototype, and contributing the changes to move the existing local disk writers behind the new API. Also thanks to Imran Rashid and Marcelo Vanzin for their continuous comments and feedback throughout the project.

Background and Motivation

The shuffle step in Spark is crucial for moving data between executors during a query, but there are a few problems with the way it's currently implemented in Spark.

First of all, shuffle file storage is not resilient and there could be various situations in which a recomputation of shuffle is necessary. A recomputation is expensive because shuffle usually requires high memory utilization and disk IO. For example, if an executor goes down during a shuffle stage, all the shuffle files that it wrote will need to be recomputed.

When running Spark on Kubernetes, the assumptions required by the External Shuffle Service do not hold. The external shuffle service in YARN provides executors' shuffle files even after the executors goes down, thus not requiring all the mapper executors to be online to complete shuffle transfers. However, in Kubernetes and other containerized environments, containers cannot necessarily access each others' disks, so shuffle files are lost with a lost executor. This once again forces shuffle files to be recomputed and wastes work.

To optimize Spark performance, system administrators also wish to reorganize their hardware configurations to isolate hosts that are optimized for compute and hosts that are optimized for disk storage. There have been endeavors in the software industry to move towards a disaggregated deployment structure with separate compute and storage clusters, so each one can be optimized for its own respective tasks more cheaply than acquiring a single host with the fully optimized compute and storage specs. To make this deployment infrastructure possible, Spark would need to write its shuffle files to an external location.

The Spark code currently ships with an interface for shuffle called the ShuffleManager. However, the implementation of this API includes the entire logic for partitioning, sorting, and aggregating data that we are not concerned with. For resiliency, performance, and containerization, the most crucial thing we need is the ability to store shuffle files in locations that are not the executors'

local disk. Currently, changing the storage behavior by extending this interface would require a lot of copy-and-pasted code, and doesn't provide other hooks that would be necessary to support remote storage (i.e. failure handling at the scheduler, location bookkeeping on the driver).

The initial proposal was to write an implementation of external shuffle that ships directly with Spark. However, after discussion with members of the community interested in this space, we discovered that people have different hardware constraints and use cases, and there was no clear solution that could easily work for everybody. Furthermore, shuffle often comprises the majority of a job's time, so having the option to optimize for a specific cluster deployment is useful. In fact, people have already forked Spark to work with other external storage systems, so we believe that having a unified way of doing so would benefit the community.

Therefore, in this SPIP, we provide a proposal for pluggable reading and writing of shuffle bytes.

Goals

- Define an API within the current shuffle implementation for pluggable writing and reading of shuffle bytes
- Determine the usefulness and implications of the API, and how it fits in with the ongoing and future work in the area of Spark shuffle

Non-Goals

- Define details of possible implementations of this API
- Discuss other possible solutions to shuffle performance and resiliency, like the feasibility of a cluster dedicated to shuffle

Target Personas

- Developers familiar with the core Spark infrastructure code, especially the code or designs behind the current implementation of shuffle
- Developers interested in improving shuffle resiliency in their own clusters by writing an external shuffle plugin implementation

Project Overview

As mentioned previously, the proposal is to introduce an API for reading and writing shuffle data from Spark's `SortShuffleManager`. The rest of the document is dedicated to defining the API and the progress that was made to reach these proposed changes.

Exploratory and background work

We started this project with https://issues.apache.org/jira/browse/SPARK-25299. After two months of prototyping a possible solution, we decided to pivot to propose an API first before writing a full implementation. The background for the decision and a detailed description of the initial API proposal is delineated here: https://docs.google.com/document/d/1NQW1XgJ6bwktjq5iPyxnvasV9g-XsauiRRayQcGLiik/edit#heading=h.y41oeu2l8y5u

The API In context of current work

The API proposed here sits behind the existing SortShuffleManager implementation of the ShuffleManager interface for writing and reading only the shuffle task's output data. It does not yet concern itself with the temporary spill files, which we'll address later in this section. At a high level, the interface for writing these files is through OutputStreams or WritableByteChannels, while the interface for reading is an InputStream. The writer returns an optional BlockManagerId if the location of the shuffle block needs to be stored by the driver (not every implementation will need to do this, we discuss this more later).

Because the API is only concerned with writing the final shuffle files, the API cannot currently support a fully disaggregated cluster where the compute cluster has little to no disk. However, if there is a concrete need to support this in the wider community, we can easily add APIs for writing the temporary spill files as well. Spill file management is orthogonal to the work we propose here.

Since the API is used from within the SortShuffleManager implementation, other implementations of the ShuffleManager will likely not benefit from this design. For example, Facebook's Cosco work that relies on a completely external shuffle cluster (where the mapper only partitions and streams its data to the appropriate shuffle cluster node and the shuffle cluster nodes take care of the sorting and aggregation work originally done by the mapper) uses a custom implementation of the ShuffleManager and thus will not hit the codepath that uses the API we plan to introduce. Similarly, Uber's shuffle manager work also extends the ShuffleManager interface to store data on remote servers instead of local disk. It is noteworthy that our work does not conflict with this previous work. Rather, the projects can be done in parallel: our work allows for flexible storage options with the existing SortShuffleManager without modifying the shuffle algorithm, while they are writing new implementations of the ShuffleManager interface to implement their own shuffle algorithm.

Our work shares many similarities to the Splash Shuffle Manager proposed by MemVerge. Both implementations aim to allow for pluggable implementations of storing temporary data. There are several significant differences between our API and the Splash shuffle manager:

1. **Code structure: Embedding in existing code vs. creating separate code paths.** Our code hooks into the existing shuffle manager, without creating a separate implementation of ShuffleManager . By contrast, the Splash shuffle manager extends the ShuffleManager API, but rewrites the sort-based shuffle algorithm, in a sense thus cloning the existing shuffle code. The risk of cloning work this way is that the upstream SortShuffleManager could diverge from the implementation of Splash's equivalent code.
2. **Shuffle index files as part of the API vs. an implementation detail.** The Splash shuffle manager has APIs for explicitly writing index files to look up partition blocks from map output files. We make the API more generic to make the implementation decide how to look up partitions after they have been written. We will discuss this further below.
3. **Splash considers map outputs to be written as files, while we consider partitions to be stored as arbitrary byte streams.** This is perhaps more so a naming convention, but Splash's APIs consider partition data to be stored in files, and as such have named the APIs using file system-based concepts. Our APIs are more generic, and as such we open the possibility for shuffle data to be stored in other kinds of data storage systems, such as NoSQL databases and distributed key-value stores.
4. **Splash handles spill files, but this is considered future work in this proposal.** Splash has an additional API for writing shuffle spill files to remote storage. We do not include this in our first incarnation of this work, but we think that it would be possible to either port over Splash's spill file API, or propose our own spill file APIs, into the Spark codebase.

We observe, then, that the APIs proposed here are strictly more generic than the file APIs given in the Splash shuffle manager. Or to put it another way, all implementations of the Splash APIs can be reworked to be put behind the proposed APIs here, and the resulting performance should be equivalent.

Some people are also using an implementation of ShuffleManager that performs the same logic as the current SortShuffleManager but also asynchronously uploads a backup copy of the shuffle files to the distributed cache Alluxio, and has logic to retrieve from that backup location when the initial fetch request from the executor fails. The new API is designed to support such an async backup implementation, where the shuffle writers could return a BlockManagerId indicating the local executor location, but still have the shuffle reader read from a DFS backup location if a read from the local executor fails.

Palantir has also been working on an implementation of external shuffle using Apache Ignite, a distributed in-memory cache. The idea combines the fault tolerance of a distributed file system with MemVerge's idea of using an in-memory store to avoid the time it takes to write bytes to disk. (link)

Finally, Palantir has also explored a default external shuffle service implementation that could ship with Spark that does not mandate the use of any external libraries, making it easy to deploy. The implementation involves individual servers that do not communicate or know about each other. In such situations, the metadata about the primary and backup locations of each shuffle block is assigned by the executor and stored within the driver. (link)

Success Criteria: Target Shuffle Storage Implementations

The objective of this API is to enable users to write shuffle storage implementations that can have all of the following properties:

1. The temporary output from map tasks can be stored resiliently. The failure of any single process in the system does not necessarily require the map output to be recomputed.
2. The temporary output from map tasks does not have to be stored on the executor nodes.
3. The implementation can be used in Kubernetes, thus enabling the user to turn on dynamic allocation in this environment.

Note that any given implementation does not need to satisfy all three of these properties. Rather, the goal is that the API would enable any implementation to have all three of these properties.

We will introduce the API into the Spark code base by moving the existing local file shuffle storage behind this API. We have already done some work to prove that this existing implementation can live behind the API (linked in PRs below).

Concretely, we considered four possible shuffle storage strategies that should be possible with this API. The criteria for success, then, is that one can reason about how the proposed API would be extended to support all of the following four strategies. These four strategies are as follows:

- The writing of shuffle files to local disk. This is the current implementation of Spark's shuffle. As mentioned previously, we will place Spark's current shuffle storage code behind the API so it can be easily replaced with a plugin implementation without the need to fork many codepaths in the Spark codebase.
- Asynchronous backup to a distributed file system. The Alluxio implementation of shuffle is an example of this. In these implementations, the shuffle files are still being stored to the executor disk. A thread asynchronous from the task worker thread will back up the files to an external storage system. The reader can later retrieve the file contents from either the original executor that wrote the files or the external storage system.
- Storing files in a distributed cache or file system. The Splash Shuffle Manager, HDFS implementation, or using a distributed cache like Ignite are all examples of external storage systems in this category. Here, the final shuffle files are written directly to a distributed file system or cache that manages its own replication.

- Individual file servers without backup or replication. In such implementations, a cluster of individual file servers store the shuffle files. The servers don't gossip, so there is no replication and no server knows about the others. Here, the driver must store the location host and port location of each shuffle block. More work, especially that related to FetchFailure integration with the scheduler, is necessary to support file system servers that handle replication.

In the next sections, we will describe some API decisions that we made in order to support all 4 types of implementations above.

Risks

- The API changes require changes to the core shuffle code itself, which could pose as a risk since it will likely affect almost every Spark query. We have attempted to mitigate these risks by running existing tests, writing microbenchmark tests, and cluster performance tests
- There is the risk that, once we start writing implementations of the plugin, there might need to be changes to the initial API. Discussing how the API could handle anticipated implementations will decrease the number and severity of changes we'd need to make later. Also, we are placing the current implementation of local shuffle file storage under this API as an initial proof of concept for both the usability and versatility of the API.
- The current design only deals with writing the final shuffle files to external locations. Temporary spills will still require disk, so fully disaggregated cluster deployments are not possible with this design alone. We would need further APIs to put spill files on remote storage.

Timeline

We have the majority of the code already written or in pull requests to this branch: https://github.com/palantir/spark/tree/spark-25299. It will likely take a couple of months to propose and merge these changes to Apache/spark, and write integration tests for the API.

We are also working on implementations of this API that involve remote storage, specifically allowing more fault-tolerant behavior and isolation when running Spark on Kubernetes. Those implementations, along with the default implementation, should be performance tested.

API Deep Dive

The API we propose comes in 5 parts: the driver lifecycle, executor lifecycle, the shuffle writer, shuffle locations metadata, and the shuffle reader.

Our initial plan is to refactor the existing shuffle logic of writing shuffle files locally behind this API. This allows us to both define all shuffle logic behind this API without having to fork the

shuffle logic, while also providing a way to test plugin implementations without being concerned about negatively impacting the current shuffle storage implementation.

We've already started writing the code for introducing the API and refactoring the current shuffle code on this branch: https://github.com/palantir/spark/tree/spark-25299. Ongoing work is also in pull requests in this fork tagged with [SPARK-25299].

We wanted to provide an API that was both flexible enough for most feasible implementations, but had a minimal surface area that obviated the need to understand how sort-based shuffle and the scheduler behave internally. Some more advanced features can be hidden by default implementations.

Driver lifecycle

```
interface ShuffleDataIO {
   ShuffleDriverComponents driver();
   ShuffleExecutorComponents executor();
}

interface ShuffleDriverComponents {
   Map<String, String> initializeApplication();
   void cleanupApplication();
   void removeShuffleData(int shuffleId);
   default boolean shouldUnregisterOutputOnHostOnFetchFailure() {
       return false;
   }
}
```

Ongoing PR: https://github.com/palantir/spark/pull/533

`ShuffleDataIO` is the entry point to the entire shuffle plugin system, and it will be reflectively instantiated from the class indicated by the configuration `spark.shuffle.io.plugin.class` (see code here). From there, `ShuffleDataIO#driver()` would be called to initialize the driver-specific parts of the plugin tree.

The driver side of the plugin allows for the following operations:

- `initializeApplication()` is a generic method for the plugin system to initialize application-wide state. For example, this can register the application with an external shuffle service process. It returns a map of additional spark configurations that will be sent to all of the application's executors.

- `cleanupApplication()` will be called upon the exit of the application, and can clean up any ephemeral state created by the shuffle plugin system. For example, this method can make calls to an external shuffle service to delete any files associated with the application.
- `removeShuffleData(int shuffleId)` is used by the ContextCleaner to delete map output data from shuffle steps that are no longer referenced.
- `shouldUnregisterOutputOnHostOnFetchFailure()` determines whether other shuffle blocks on the same host as a block that resulted in a `FetchFailureException` should be removed. For example, in the case where blocks are backed up asynchronously (i.e. executors store the shuffle files locally and then upload them to a DFS in the background), one wouldn't want to remove all the blocks on the same host as a unfetchable block. In that case, perhaps the executor died while uploading a backup, but other backups by the same executor are still persisted in the DFS, and other blocks written by other executors on the same host might still be serviceable. Therefore, in such an implementation, `shouldUnregisterOutputOnHostOnFetchFailure()` should return false. However, in the individual file server case, the most likely scenario for a fetch failure would be the file server being unresponsive or going down, in which case one should trigger a recomputation of everything stored on that host, and thus `shouldUnregisterOutputOnHostOnFetchFailure()` should return true.

Executor Components

```
interface ShuffleExecutorComponents {
   void initializeExecutor(
      String appId,
      String execId,
      Map<String, String> extraConfigs);

   ShuffleWriteSupport write();

   ShuffleReadSupport read();
}
```

Reference PR section:
https://github.com/palantir/spark/commit/bc40da2a765ae38de107bcf74386ac23463d91d1#diff-f0a98bdcfed7b93ab277e2b92c8fd9ecR216

Executor components are initiated in the `SortShuffleManager` and are responsible for constructing the writers and readers. If the executor needs to do initialization tasks (i.e. register with the shuffle service), then it can do so in `initializeExecutor()`. It's also passed the extra configurations set in `ShuffleDriverComponents.initializeApplication()`.

Shuffle Writer

```
public interface ShuffleWriteSupport {
   ShuffleMapOutputWriter createMapOutputWriter(
       int shuffleId,
       int mapId,
       int numPartitions) throws IOException;
}

public interface ShuffleMapOutputWriter {
    ShufflePartitionWriter getPartitionWriter(int partitionId)
       throws IOException;

    Optional<BlockManagerId> commitAllPartitions()
       throws IOException;

    void abort(Throwable error) throws IOException;
}

public interface ShufflePartitionWriter {

   /**
    * Returns an underlying {@link OutputStream} that can write bytes
    * to the underlying data store.
    * <p>
    * Note that this stream itself is not closed by the caller; close
    * the stream in the implementation of this interface's
    * {@link #close()}.
    */
   OutputStream toStream() throws IOException;

   /**
    * Get the number of bytes written by this writer's stream returned
    * by {@link #toStream()} or the channel returned by
    * {@link #toChannel()}.
    */
   long getNumBytesWritten();
}
```

Reference PRs:

- https://github.com/palantir/spark/pull/524

- https://github.com/palantir/spark/pull/540
- https://github.com/palantir/spark/pull/535

On the writer side, there is a single `ShuffleMapOutputWriter` for each map task. Each mapper creates a single `ShufflePartitionWriter` for a single partition. Each partition writer returns an `OutputStream` that the shuffle implementation can write its shuffle data to.

If the map task fails for any reason, `ShuffleMapOutputWriter#abort` is called to revert any partial work that was done by the map output writer or any of its partition writers.

When the map task is completed, `ShuffleMapOutputWriter#commitAllPartitions` is called to finalize the writing from this map task. The commit returns metadata about where the shuffle data was persisted as an `Optional<BlockManagerId>`. The return value is Optional.empty for DFS use cases since the host/port of the remote store can be configurable. For async and local file implementations, this would be the executor from which to retrieve the data. For remote file servers, the BlockManagerId would include the host and port of the remote file server, with a null execId since the remote file server isn't running an executor.

There is an additional API that users can optionally implement if they want their implementation to support `WritableByteChannels` to transfer data to the shuffle locations:

```
public interface SupportsTransferTo extends ShufflePartitionWriter {

   /**
    * Opens and returns a {@link TransferrableWritableByteChannel} for
    * transferring bytes from partial input byte channels to the
    * underlying shuffle data store.
    */
   TransferrableWritableByteChannel openTransferrableChannel()
       throws IOException;

   /**
    * Returns the number of bytes written either by this writer's
    * output stream opened by
    * {@link #openStream()} or the byte channel opened by
    * {@link #openTransferrableChannel()}.
    */
   @Override
   long getNumBytesWritten();
}

public interface TransferrableWritableByteChannel extends Closeable {
```

```
    /**
     * Copy all bytes from the source readable byte channel into this
     * byte channel.
     * @param source File to transfer bytes from. Do not call anything
     *        on this channel other than {@link
     *        FileChannel#transferTo(long, long, WritableByteChannel)}.
     * @param transferStartPosition Start position of the input file to
     *        transfer from.
     * @param numBytesToTransfer Number of bytes to transfer from the
     *        given source.
     */
    void transferFrom(
        FileChannel source,
        long transferStartPosition,
        long numBytesToTransfer) throws IOException;
}
```

The `WritableByteChannel` implementation is necessary for the implementation of the writer that writes to local disk because it allows the write to bypass memory if we're copying from one file to another. For implementations where a `WritableByteChannel` doesn't make sense, a default implementation to convert from stream to channel is provided so the implementer doesn't need to be concerned with channels.

Wrapping the input and output `WritableByteChannels` inside of `TransferrableWritableByteChannel` is necessary to be able to override the `close()` method on the `WritableByteChannel`. In the local disk implementation of shuffle, each partition writer actually returns the same stream since all partitions are written to the same file, so we don't want to close the `OutputStream` until the very end, when we commit. However, not all implementations behave this way, and some may need to close every partition's `OutputStream` separately. Therefore, to avoid the risk of leaking resources, we need to close every `OutputStream`, so we override the local disk implementation's `close()` method to not close until we commit the entire partition.

The same logic applies to `WritableByteChannel`. The optimizations for transferring bytes bypassing memory are only effective when transferring from `FileChannel` to `FileChannel`, so we would need to return an instance of `FileChannel`, which is constructed as part of the `FileOutputStream` object. To override the close functionality, we would need to extend the `FileChannelImpl` class, which seems dubious since the NIO classes are provided by the JDK. Therefore, we instead return a `TransferrableWritableByteChannel` whose `close()` method is easily overridable. Thus, in the local disk implementation, we will just choose not to call `close()` on the underlying `WritableByteChannel` whenever `close()` is called.

AttemptId in MapStatus

PR: https://github.com/palantir/spark/pull/574

The Spark scheduler supports speculative execution, meaning that two executors could be running the same task at the same time, and stage retries, meaning that the same task could be running in an active taskset as well as a zombie taskset. We need to ensure that, in both cases, the two executors don't override each other's data, and that a shuffle reader reads accurate data. For example, take an instance where the writer implementation writes each partition to the remote host in a sequence of chunks. In such a situation, a reducer might read data half written by the original task and half written by the running speculative task, which will not be the correct contents if the mapper output is unordered. Therefore, writes by a single mapper might have to be transactional, which is not clear from the API, and seems rather complex to reason about, so we shouldn't expect this from the implementer.

We introduce the idea of an attempt Id in both the API and the `MapStatus` structure to ensure that readers always read consistent data.

The attempt id could also be helpful towards https://issues.apache.org/jira/browse/SPARK-25341 by acting as a "shuffle generation number," where the reducer should specify which shuffle it wants to read (as mentioned in the ticket, this is necessary for rolling back non-deterministic shuffle stages).

Shuffle Reader

```
public interface ShuffleReadSupport {
   /**
    * Returns an underlying {@link Iterable<InputStream>} that will
    * iterate through shuffle data, given an iterable for the shuffle
    * blocks to fetch.
    */
   Iterable<InputStream> getPartitionReaders(
       Iterable<ShuffleBlockInfo> blockMetadata) throws IOException;
}

public class ShuffleBlockInfo {
   public final int shuffleId;
   public final int mapId;
   public final int reduceId;
   public final long length;
   Public final int attemptId;
   public final Optional<BlockManagerId> shuffleLocation;
```

```
}
```

Reference PR: https://github.com/palantir/spark/pull/523

On the reader side, the `ShuffleReadSupport` is passed an iterable over `ShuffleBlockInfo` that contains the information necessary to retrieve each shuffle block, and returns an iterable over `InputStreams`. This flexible API allows the implementation to decide how to fetch its bytes. For example, the current Spark shuffle code pre-fetches files onto disk before the files' `InputStream` is returned by the iterator.

Failure Handling

Currently, if a reduce job fails to retrieve a shuffle block, Spark will assume that there was an error at the remote end of the request and try to mark the storage location as unavailable. This means that, when a reducer throws a `FetchFailedException`, the DAGScheduler will mark all data on that executor or host as unavailable. (if the external shuffle service is disabled, Spark tries to fetch from the mapper executor directly, so it only marks data on the executor as unavailable. However, if the external shuffle service is enabled, then Spark can mark all data on the host as unavailable since the external shuffle service is supposed to service all shuffle data on the host). Marking mapper data as unavailable allows the scheduler to retry running the mapper stage. Otherwise, the entire job will fail.

When using remote storage, we don't necessarily want Spark to blacklist like it did before. Here, we outline how we can handle fetch failures in each of the 4 implementations in a V1 implementation of this API:

1. The writing of shuffle files to local disk. This is handled the same way as it's currently done in Spark since it's the same implementation. Here, the reducer throws a `FetchFailedException` by passing in the remote host and port that it was trying to read from. The scheduler then blacklists that host, or the host-port combination (for blacklisting by executor id instead of host).
2. Writing shuffle data to local disk and then asynchronously backing up the data to a resilient remote storage layer. Here, the reducer will try to read from both a mapper executor (to retrieve its local file) and the remote store where the backup is stored. If the read from both fails, we can throw a `FetchFailedException` with the BlockManagerId of the executor. Then, the scheduler will retrigger the recomputation of any shuffle block stored on the executor. Note that this could potentially rerun map tasks that have been already backed up, but it's better to be pessimistic when rerunning mapper tasks because Spark will abort the entire job if too many task sets fail to complete.
3. Synchronously storing the shuffle data in a distributed cache or file system. When using a distributed storage layer for storing shuffle files, there should ideally never be failures in fetching shuffle data. To indicate that nothing should be recomputed, the plugin can

throw a generic Exception that is not a `FetchFailedException`. However, in the off-chance that the plugin wants to report a missing output, it should throw a `FetchFailedExecption` with a null BlockManagerId.

4. Writing shuffle data to individual independent file servers. Here, if the reducer fails to fetch the data from a remote host, it will assume that the host is unreachable and all data on that host is lost. Then, the plugin can trigger a recomputation of all shuffle blocks on a file server by returning the BlockManagerId of the remote server (with a null execId) with the FetchFailedException.

First API Iteration Limitations

- We don't support individual independent file servers with backup locations. We experimented with a `ShuffleLocation` abstraction for denoting backup locations to the MapOutputTracker (see PR https://github.com/palantir/spark/pull/517/files), but integration with the scheduler was complex (attempts: https://github.com/palantir/spark/pull/548, https://github.com/palantir/spark/pull/555, https://github.com/palantir/spark/pull/559). We decided not to support this use case in the first iteration, and address the issue if it becomes an important concern later.
- In the async backup case (implementation #2 above), there's no way to guarantee persistence of a shuffle block if the executor fails for an unexpected reason or in the case of dynamic allocation. Ideally, the executor would upload all active shuffle blocks to the remote location before shutting down. Additional plugin points might be needed for communication between driver and the executor to optimize this case. Without this guarantee, the async implementation does offer a large improvement over the existing shuffle behavior.