Shuffle Metadata Tracking Design

Background

In Spark, there has been an ongoing discussion regarding the need for a wider array of extension options for Spark's shuffle logic. The overarching goal is to support storing shuffled data in other systems, or to offload the shuffle algorithm to another system entirely. We remark that here we are narrowly focused on the mandate from SPARK-25299, which only deals with storing shuffle files in external mediums but otherwise leaves the shuffle algorithm intact (e.g. the Spark executors in the applications are themselves still shuffling the data, possibly spilling to local disk).

We will not discuss the entirety of the discourse that has occurred so far. Before going further, one is encouraged to follow the discussions that have occurred previously, particularly in the following forums:

- SPARK-25299 Documentation
    - The base JIRA: https://issues.apache.org/jira/browse/SPARK-25299
    - The SPIP

- This e-mail thread on the dev list outlining various proposed features to Spark's shuffle logic
- The MapStatus Discussion Design Document

The Need For Shuffle Metadata Tracking

A common discussion topic in all of these forums has been around how we track the shuffle data that is stored in these external systems. For example, consider the following shuffle storage implementations:

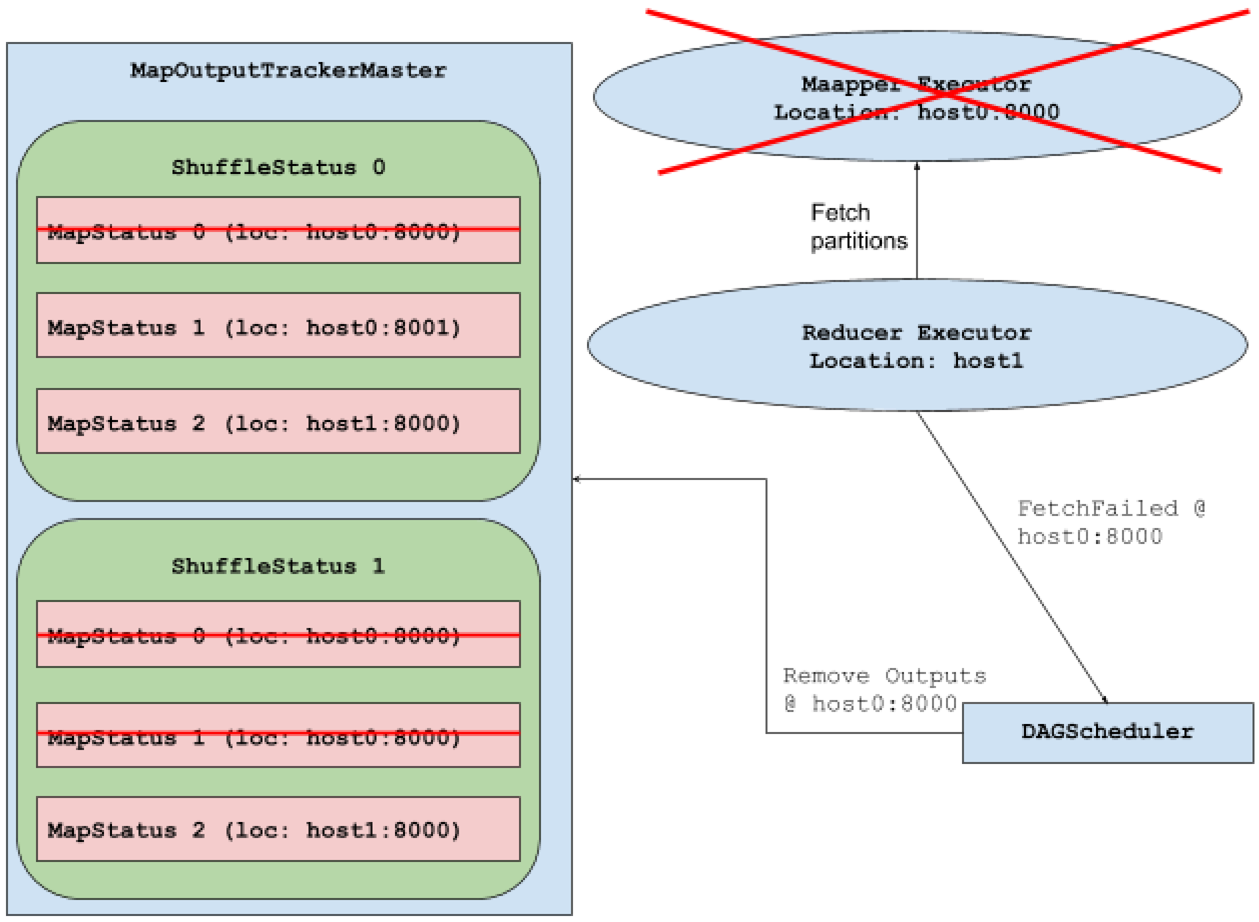
1. Data is replicated to multiple individual file servers, and at read time we need to know the URI for a given block.
2. Data is stored in a distributed file system and we need to know the path to pass to a Hadoop `FileSystem#open` call
3. Data is stored on executors and is asynchronously replicated to an external storage medium - and we need to know, at read time, if the data is available in the external storage system.

We can see that in some of these cases it would be trivial for the external storage system itself to provide the information that is required to read the shuffle data. For example, if data was only stored in a distributed file system and is fetched via `FileSystem#open(URI)`, the reader could just fetch the file located at a URI of the format `hdfs:///tmp/<app-id>/<shuffle-id>-<map-id>-<attempt-id>` - this is in fact similar to how Spark itself locates shuffle files on the local disks of executors. But in other cases such as (1) and (3) above it is evident that some metadata would need to be tracked in order for the shuffle readers to successfully fetch the data in question.

Fetch Failure Handling in Hybrid Shuffle Storage Solutions

Complications further arise when considering hybrid storage solutions that keep a portion of the shuffle data on executors and some of the shuffle data in remote storage. In such architectures, questions arise around how to determine when shuffle data is lost when a reader fails to fetch shuffle data. We henceforth refer to this class of error as a **Fetch Failure** as the Spark nomenclature does.

Currently, in the event of a task failing due to a fetch failure, Spark assumes that all shuffle data generated by that executor is completely lost and needs to be recomputed. In the case that the external shuffle service is being used, a failure to fetch from the external shuffle service is assumed to be a failure of the entire host itself. The external shuffle service is a single process on each node - thus a failure to fetch any blocks from that external shuffle service means that all shuffle data written by any executor that was on that shuffle service's node is lost.


MapOutputTrackerMaster
ShuffleStatus 0
~~MapStatus 0 (loc: host0:8000)~~
MapStatus 1 (loc: host0:8001)
MapStatus 2 (loc: host1:8000)
ShuffleStatus 1
~~MapStatus 0 (loc: host0:8000)~~
~~MapStatus 1 (loc: host0:8000)~~
MapStatus 2 (loc: host1:8000)
~~Maapper Executor Location: host0:8000~~
Fetch partitions
Reducer Executor Location: host1
FetchFailed @ host0:8000
Remove Outputs @ host0:8000
DAGScheduler

Above: A fetch failure from the mapper executor, results in the assumed loss of all map outputs on that executor. The lost map outputs must be recomputed. But what if they were stored in an external system?

Fetch failures should be less likely in cases where shuffle writers store their data entirely in external systems that are resilient. However, storage of shuffle files in lower availability systems, such as individual non-replicated file servers, need a way to inform Spark's task scheduler that blocks are lost and need to be recomputed. Furthermore, in the case that blocks can be stored in external systems, fetch failures do not necessarily imply the loss of all blocks that were written by the mapper executor. This behavior also applies in the case where some blocks are stored on the executors, but other blocks are also replicated in remote storage.

Current Behavior and Architecture

Before going further, it helps to define terms that will be used throughout the document, and to give a brief summary of the existing behavior and architecture of task scheduling in Spark.

Definitions

The following modules are relevant for our current discourse:

- **MapStatus**: A lightweight structure representing the output of a completed map task. It holds the identifier of the executor that ran this map task, as well as the lengths of each partition written by this task. `MapStatus` objects currently represent the core premise that the outputs of map tasks are written on the local disks of the executors that ran those map tasks.
- **ShuffleStatus**: A structure representing a collection of map outputs (MapStatuses) for a given shuffle stage.
- **MapOutputTrackerMaster**: Module on the driver responsible for tracking available map outputs for each stage. Mostly wraps a collection of `ShuffleStatus` objects, one per shuffle stage performed by the Spark application. Contains functions to add map outputs when map tasks complete, as well as remove map outputs due to fetch failures or other conditions.
- **MapOutputTrackerWorker**: Module instantiated on each executor that represents a hook to the driver's corresponding `MapOutputTrackerMaster`. Generally delegates to calling the `MapOutputTrackerMaster` so that map output metadata (`MapStatus`es) can be accessed for shuffle read and write operations.
- **DAGScheduler**: Large composite module on the driver responsible for coordinating task scheduling and task completion handling. This module is particularly complex and isn't easily summarized here - but we'll generally be narrowly focused to the scheduler's successful map task completion handling as well as its fetch failure handling.

The module structures that are easily summarized and are critical to our discussion are given in pseudo-code below - note that all of the pseudo-code we'll be providing through this discourse isn't 100% accurate, but gives a reasonable summary of the way shuffle metadata is represented in Spark:

```
struct BlockManagerId {
  String executorId
  String host
  int port
}

struct MapStatus {
  BlockManagerId location
  long[] partitionLengths
}

struct ShuffleStatus {
  MapStatus[] mapStatuses; // 1 MapStatus per map task in this stage
}

class MapOutputTrackerMaster {
  Map<Integer, ShuffleStatus> shuffleStatuses // Keyed by shuffle id
  // Functions omitted
}

class DAGScheduler {
  // Has a hook on the MapOutputTrackerMaster
  MapOutputTrackerMaster outputTrackerMaster

  // Functions omitted
}
```

Map Task Completion Process

When a map task completes, the executor sends the driver a task completion event. Map tasks send back a `MapStatus` object containing the executor location where that task ran, and the lengths of the partitions written in the map output file. Upon reception of the event, the `MapStatus` object is stored in the `MapOutputTrackerMaster`. We can summarize this as follows:

```
// Run on executors but has a hook to the driver
class MapTask {
  DAGScheduler driverDagScheduler
  BlockManagerId location
  int shuffleId
  int mapId

  void runTask(inputs: Iterator<Object>) {
    Iterator<Object> outputs <- compute(inputs)
    // partitionsBytes[i] are the bytes for partition ‘i’
    // Abstract away sorting and partitioning
    byte[][] partitionsBytes <- partitionAndSerialize(outputs)
    Vector<Long> lengths = new Vector<Long>()
    FileOutputStream partOutput <- openDataFile()
    FileOutputStream indexOutput <- openIndexFile()
    for partition in partitionsBytes {
      partOutput.write(partition)
      indexOutput.write(partition.length)
      lengths.push(partition.length)
    }
    partOutput.close()
    indexOutput.close()
    MapStatus status <- new MapStatus(location, lengths.toArray())
    driverDagScheduler.onMapTaskCompletion(
        shuffleId, mapid, mapStatus)
  }
}

class DAGScheduler {
  def onMapTaskCompletion(
      int shuffleId, int mapId, MapStatus mapStatus) {
    // Does other things, but for our purposes this is all that is
    // relevant
    outputTrackerMaster.registerMapOutput(
        shuffleId, mapId, mapStatus)
  }
}

class MapOutputTrackerMaster {
  def registerMapOutput(
      int shuffleId, int mapId, MapStatus mapStatus) {
    shuffleStatuses.get(shuffleId).mapStatuses[mapId] <- mapStatus
  }
}
```

(A visual for the above is also provided in a diagram in this document)

Fetch Failure Handling

When an executor fails to read shuffle data from another executor or a node's shuffle service during a reduce task, the `DAGScheduler` first instructs the `MapOutputTracker` to remove all of the map outputs that were stored ono that executor. Then, the scheduler has to resubmit any tasks from the map stage and its dependent stages where the map outputs were lost.

Removing Map Outputs

First, here's a summary of how fetch failures cause map outputs to be removed from the map output tracker:

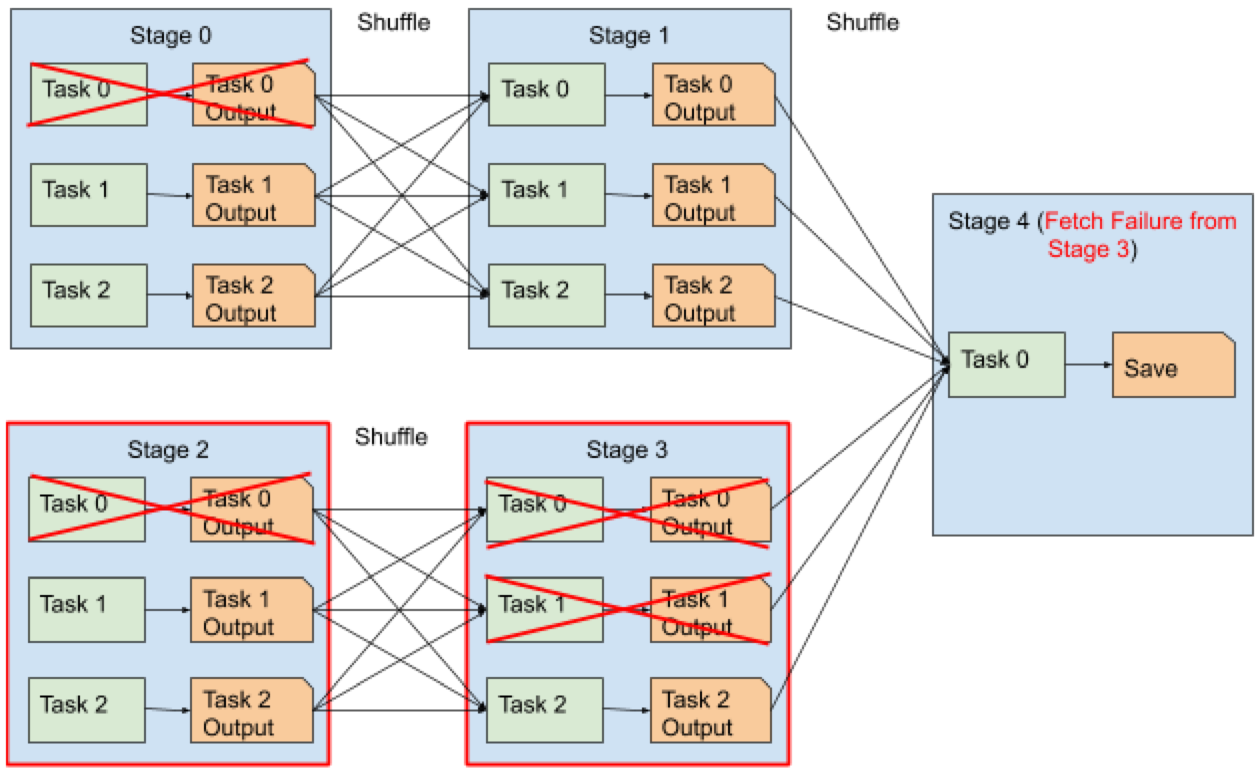
```
class DAGScheduler {
  void handleFetchFailure(
      int shuffleId, int mapId, BlockManagerId failedMapper) {
    // other logic omitted...
    if external shuffle service is enabled &&
        App is configured to unregister outputs on hosts {
      outputTrackerMaster.removeOutputsOnHost(failedMapper.host)
    } else {
      outputTrackerMaster.removeOutputsOnExecutor(failedMapper)
    }
  }
}

class MapOutputTrackerMaster {
  void removeOutputsOnHost(String host) {
    for shuf in shuffleStatuses.values {
      for i in 0 until shuf.mapStatuses.length {
        MapStatus mapStatus <- shuf.mapStatuses[i]
        if mapStatus.location.host == host {
          mapStatuses[i] <- null
        }
      }
    }
  }

  void removeOutputsOnExecutor(BlockManagerId executor) {
    for shuf in shuffleStatuses.values {
      for i in 0 until shuf.mapStatuses.length {
        MapStatus mapStatus <- shuf.mapStatuses[i]
        if mapStatus.location == executor {
          mapStatuses[i] <- null
        }
      }
    }
  }
}
```

The gist of the above code is to iterate through all the `MapStatus` objects currently held by the `MapOutputTrackerMaster`, and remove all the ones that were on the same location as the location that failed to provide shuffle data to the reducer. We briefly remark that if the external shuffle service is enabled, the user can choose to force all map outputs on the shuffle service's host to be recomputed upon a fetch failure as well. A `null MapStatus` in the `mapStatuses` array of a given `ShuffleStatus` indicates that there does not exist a location where the map task's output can be fetched from - or to put it another way, it's as if the map task had never executed in the first place. Thus if the output of this map task is needed in the Spark job for downstream reduce stages, these tasks would need to be re-executed.

Resubmitting Failed Stages

When a fetch failure occurs, it is required to re-run the tasks that were lost from the map side of that shuffle. However, there are cases where more stages would need to be re-submitted as well.

Stages in Spark form a directed acyclic graph, where each shuffle stage has a set of input stages. A shuffle stage needs to be recomputed if any of its map outputs are missing. But in a shuffle, a shuffle stage cannot be executed unless all of its input stages have all of their map outputs present. The result is that in the case of a fetch failure, stage retries may cascade through multiple layers of the stage graph.



Above: A stage graph with multiple shuffles. Tasks with red strikethroughs have lost output files because of a fetch failure on an executor. Stages with red boxes will have all tasks with red strikethroughs through them re-executed. Therefore, the full list of stages with re-executed tasks will be: (Stage 2, [Task 0]), (Stage 3, [Task 0, Task 1]). Note we do not re-execute any tasks from stage 0, even though tasks are lost, since we have all the outputs we need to complete stage 4 from stage 1.

Finding stages and tasks that need to be retried is a depth-first search through the stage graph, starting from the stage where the fetch failure occurred, and working upwards through the parents. A branch of the search tree is cut short if the current stage node of the graph has all of its map outputs present.

For brevity, we defer the inclusion of the pseudo-code for the algorithm to Appendix A.

Shuffle Output Tracking API

Given the existing architecture, and the requirement to avoid recomputing shuffle blocks that are stored externally, we now turn to the matter of designing an API that can satisfy the requirements. Since task scheduling is done on the driver, and shuffle metadata should be persisted throughout the lifetime of the application, it makes sense to start with a singleton module on the driver, loaded via SPARK-25299's `ShuffleDriverComponents`.

```
class ShuffleDriverComponents {

  // Shuffle metadata support is optional
  Optional<ShuffleOutputTracker> shuffleOutputTracker()
}

class ShuffleOutputTracker {
  // To be filled in
}
```

Metadata Tracking

In order for readers to receive additional information about the storage of shuffle blocks, we first enable custom map output writers to return metadata to the driver. The shuffle output tracker can then be told about the completion of the map task and pass along the metadata that was returned by the map output writer. Metadata must be serializable so that it can be transmitted from the executors to the driver.

```
struct MapOutputWriterCommitMessage {
  long[] partitionLengths
  Optional<MapOutputMetadata> mapOutputMetadata
}

interface MapOutputMetadata extends Serializable {}

public interface ShuffleMapOutputWriter {

  // See SPARK-25299
  ShufflePartitionWriter getPartitionWriter(int partitionId)

  MapOutputWriterCommitMessage commitAllPartitions()

  void abort(Throwable error) throws IOException
}

class ShuffleOutputTracker {

  void registerShuffle(int shuffleId, int numMaps)
  void registerMapOutput(
      int shuffleId,
      int mapId,
      long mapAttemptId,
      MapOutputMetadata metadata)
}
```

A few remarks on the above API snippet:

- `MapOutputWriterCommitMessage` still encapsulates the `partitionLengths` of the written blocks, as this allows the higher-level writers to return these partition lengths back to the driver via the `MapStatus` structure.
- We include the attempt ID of the map task in the call to `ShuffleOutputTracker#registerMapOutput` so that the implementation can track metadata written by two map output writers for the same task on different attempts.

The shuffle readers need to be able to use the metadata tracked by the shuffle output tracker. However, it is not necessarily the case that the individual `MapOutputMetadata` objects that were registered via `registerMapOutput` should be given directly to the readers. As a specific example, we consider an implementation that replicates blocks between multiple file servers. A single map output metadata may indicate the file being saved on one file server - but on the read side, there might need to be more information required to actually read the files, such as secondary locations where the file has since been replicated to.

As such, we propose a more general interface, where a reader can be given a single blob of metadata pertaining to all the shuffles under the shuffle id for all the blocks that are to be read:

```
Interface ShuffleMetadata extends Serializable {}

interface ShuffleOutputTracker {

  void registerShuffle(int shuffleId, int numMaps)

  void registerMapOutput(
      int shuffleId,
      int mapId,
      long mapAttemptId,
      MapOutputMetadata metadata)

  Optional<ShuffleMetadata> getShuffleMetadata(int shuffleId)
}

struct ShuffleBlockInfo {
  int shuffleId
  int mapId
  int reduceId
  long mapTaskAttemptId
  BlockManagerId mapperLocation
}

interface ShuffleExecutorComponents {
  Iterable<ShuffleBLockInputStream> getPartitionReaders(
      Iterable<ShuffleBlockInfo> blocks,
      ShuffleDependency dependency,
      Optional<ShuffleMetadata> shuffleMetadata)
}
```

Hooking in the provision of the shuffle metadata is a matter of augmenting the code paths for the `BlockStoreShuffleReader` on the executors to fetch the appropriate shuffle metadata from the driver alongside the rest of the metadata that was being fetched previously.

Dynamic Metadata Tracking

It is important for executors to be able to dynamically inform the shuffle output tracker of updates being made to the storage of shuffle data. For example, in the case when shuffle files are being uploaded asynchronously, the executor needs to send the driver a notification when the backup of a map output is complete.

This requires two plugin points - one of the driver to receive update messages, and one on the executor to send these update messages. We remark that updates may not be tied directly to map outputs. For example, an executor may choose to back up individual blocks of a map output independently. So the APIs need to be considerably generic, as any given update message may need to pass an arbitrary payload.

The APIs on the driver's `ShuffleOutputTracker` may look as follows:

```
interface ShuffleUpdateMessage extends Serializable {}

interface ShuffleOutputTracker {
  void onShuffleUpdate(ShuffleUpdateMessage message)
}
```

On the executor side, the `ShuffleExecutorComponents` needs to have a hook to post shuffle update messages to. This hook is *not* meant to be extended by the shuffle plugin author. Instead, it will have a concrete implementation that sends the message to the driver via Spark's Netty RPC protocol. We also don't want to provide the entirety of the `ShuffleOutputTracker`'s methods to the executors, since most of the methods in

ShuffleOutputTracker are meant to be invoked in the context of the scheduler and the lifecycle of Spark tasks.

```
Interface ShuffleUpdateSender {
  void sendShuffleUpdate(ShuffleUpdateMessage message)
}

interface ShuffleExecutorComponents {
  void initializeExecutor(
      String appId,
      String executorId,
      Map<String, String> extraConfigs,
      ShuffleUpdateSender updateSender)
}
```

A couple of things we can iron out from these APIs:

- Will there be a need to provide `BlockManagerId`s in the `MapOutputWriterCommitMessage` and `ShuffleBlockInfo`? Generally we should be thinking of ways to reduce the API surface area across the board here.
- Should we guarantee that calls to the `ShuffleOutputTracker` are made in a thread-safe manner, or should implementations be aware of their internal need for concurrent access?

Avoiding Redundant Task Recomputation

In tackling the task recomputation problem, shuffle plugins need to inform the scheduler that blocks do not need to be recomputed even if a task is lost. One way to prevent blocks from being recomputed is by avoiding them being invalidated by the scheduler in the first place. Spark assumes that blocks that were written by a mapper executor are lost when that mapper executor or shuffle service is lost, so we can allow the plugin to intervene in the scheduler's logic to choose to remove the block. For example:

```
interface ShuffleOutputTracker {

  void registerShuffle(int shuffleId, int numMaps)

  void registerMapOutput(
      int shuffleId,
      int mapId,
      long mapAttemptId,
      MapOutputMetadata metadata)

  Optional<ShuffleMetadata> getShuffleMetadata(int shuffleId)
  void handleFetchFailure(
      int shuffleId,
      int mapId,
      long mapTaskAttemptId,
      Int reduceId,
      BlockManagerId mapperLocation)

  boolean isMapOutputAvailableExternally(
      int shuffleId, int mapId, long mapTaskAttemptId)
}

// Augment MapStatus with mapTaskAttemptId, which wasn't there
before.
struct MapStatus {
  BlockManagerId location
  long[] partitionLengths
  Long mapTaskAttemptId
}

class DAGScheduler {
  // Instance is shared with the MapOutputTrackerMaster
  Optional<ShuffleOutputTracker> shuffleOutputTracker

  void handleFetchFailure(
      int shuffleId,
      int mapId,
      long mapTaskAttemptId,
      Int reduceId,
      BlockManagerId failedMapper) {
    // other logic omitted…
    shuffleOutputTracker.foreach { resolvedTracker =>
      resolvedTacker.handleFetchFailure(
          shuffleId,
          mapId,
          mapTaskAttemptId,
          reduceId,
          failedMapper)
    if external shuffle service is enabled &&
        App is configured to unregister outputs on hosts {
      outputTrackerMaster.handleFetchFailureByHost(failedMapper.host)
    } else {
      outputTrackerMaster.handleFetchFailureByExecutor(failedMapper)
    }
  }
}

class MapOutputTrackerMaster {

  Optional<ShuffleOutputTracker> shuffleOutputTracker

  void handleFetchFailureByHost(String host) {
    for (shuffleId, shuf) in shuffleStatuses {
      for mapId in 0 until shuf.mapStatuses.length {
        MapStatus mapStatus <- shuf.mapStatuses[mapId]
        if mapStatus.location.host == host &&
            (shuffleOutputTracker.isEmpty ||
            !shuffleOutputTracker.isMapOutputAvailableExternally(
                shuffleId, mapId, mapStatus.mapAttemptId)) {
          mapStatuses[i] <- null
        }
      }
    }
  }

  void handleFetchFailureByExecutor(BlockManagerId executor) {
    for (shuffleId, shuf) in shuffleStatuses {
      for mapId in 0 until shuf.mapStatuses.length {
        MapStatus mapStatus <- shuf.mapStatuses[mapId]
        if mapStatus.location == executor &&
            (shuffleOUtputTracker.isEmpty ||
            !shuffleOutputTracker.isMapOutputAvailableExternally(
                shuffleId, mapId, mapStatus.mapAttemptId)) {
          mapStatuses[i] <- null
        }
      }
    }
  }
}
```

The hook for the shuffle output tracker to handle fetch failures is a concession to implementations that may use the executor’s local disk as one of the storage locations. In this case, custom shuffle reader implementations may be able to throw their own fetch failure exceptions, and these should be propagated back to the driver by this API.

Each time a map output would be removed from the map output tracker, we first check if the map output could be fetched from an external location. We thus avoid removing any such map outputs. Then, when stages are re-submitted, we will not recompute these tasks since their map outputs remain available. It's the responsibility of the shuffle reader to ensure that a subsequent read of these blocks are fetched from the external system instead of from the executor that just crashed.

Some more remarks about the above architecture:

- This maintains most of the structure of the existing scheduler and map output tracker. Another option that has been floated is to move much of the existing map output tracking logic behind the ShuffleOutputTracker API - see here. Here, we have opted for an approach that is less invasive of the existing code.
- The idea of unregistering outputs from the map output tracker, based on the host vs. based on the executor - this idea seems like an implementation detail that only matters in the existing implementation of shuffle storage on local disk. It is unclear if this behavior should be configurable by the shuffle plugin.
- Again, how much can we reduce the exposure to `BlockManagerId` in the API vs. passing it along in opaque metadata objects?
- Should Fetch Failures be capable of providing arbitrary metadata blobs to the shuffle output tracker?

Proof of Concept Code

Two proof of concepts are available - feel free to leave comments or questions ono either of them:

- An end to end implementation here. This also demonstrates how the new shuffle tracking APIs can be integrated in an implementation that asynchronously backs up shuffle files to remote storage. This implementation was briefly tested with cases where fetch failures were forced to occur - as expected, map outputs that were successfully backed up to remote storage were not recomputed when the mapper stage was re-submitted upon a fetch failure.
- An incremental set of PRs against the latest version of Spark - this is a work in progress and will be updated as we go along:
  - Returning the map output metadata from shuffle writers: https://github.com/mccheah/spark/pull/10
  - Registering map output metadata in the shuffle output tracker: https://github.com/mccheah/spark/pull/11
  - Use the shuffle metadata for reading shuffle blocks: https://github.com/mccheah/spark/pull/12
  - Avoid recomputing tasks that have outputs that are persisted externally: https://github.com/mccheah/spark/pull/13
  - Allow shuffle output metadata to be updated dynamically by the executors: https://github.com/mccheah/spark/pull/14

Discussion and Next Steps

Feel free to leave comments throughout the document to discuss further. When we have coalesced around a general design, it would be appropriate to move forward with implementing this code in open source Spark.

A rough breakdown of the tasks at hand - we can contribute these incrementally:

1. Add `ShuffleOutputTracker` API with metadata tracking methods only - `registerShuffle`, `registerMapOutput`, `getShuffleMetadata`, un-registration methods
2. Augment `MapOutputWriterCommitMessage` to include the additional optional `MapOutputMetadata` field
3. Add hooks for passing `MapOutputMetadata` from writers to the `ShuffleOutputTracker`
4. Add shuffle reader APIs which accept `ShuffleMetadata` arguments as well as the block infos.
5. Implement the shuffle reader API in local disk read mode (i.e. put existing shuffle read code behind the API)
6. Add `ShuffleOutputTracker` APIs for handling fetch failures and reporting partitions being available externally
7. Add logic to avoid removing map outputs when partitions are available externally
8. Add hook from executors to driver via `ShuffleUpdateSender`

Appendix A: Pseudocode For Finding Stages To Re-Submit

```
// Not a real thing in Spark, but helps to encapsulate a stage along
// with its missing partitions for this example.
struct MissingPartitions {
  int stageId
  List<Integer> missingPartitions
}

class MapOutputTrackerMaster {

  List<Integer> findMissingPartitions(int shuffleId) {
    List<Integer> missingPartitions = new List()
    ShuffleStatus shuf = shuffleStatuses.get(shuffleId)
    for i in 0 until shuf.mapStatuses.length {
      if shuf.mapStatuses[i] == null {
        missingPartitions.add(i)
      }
    }
    Return missingPartitions
  }
}
class DAGScheduler {

  // Assume there's some mapping between a stage id and the shuffle
id
  // of its map outputs. This isn't entirely accurate (see
  // ShuffleMapStage) but is a decent approximation for now.
  int getShuffleId(int shuffleId) {... }

  Set<Integer> getDependentStages(int stageId) { ... }

  Deque<MissingPartitions> findStagesToRecompute(int stageId) {
    Deque<MissingPartitions> failedStages = new Deque<Integer>()
    List<Integer> missingParts =
        outputTrackerMaster.findMissingPartitions(
            getShuffleId(stageId))
    if missingParts.nonEmpty {
      failedStages.push(new MissingPartitions(stageId, missingParts))
      Set<Integer> dependents = getDependentStages(stageId)
      for dependent in dependentStages {
        // Note we don't check if we're visiting something twice,
        // unfortunately - but this is more for the sake of an
        // outline…
        failedStages.addAll(findStagesToRecompute(dependent))
      }
    }
    return failedStages
  }
}
```