Adaptive Execution in Spark

Motivation

Query planning is one of the main factors in high performance, but the current Spark engine requires the execution DAG for a job to be set in advance. Even with cost-based optimization, it is hard to know the behavior of data and user-defined functions well enough to always get great execution plans. This is more true in Spark than in, say, database workloads, because Spark often runs on new data it has never had a chance to index, and runs many UDFs in addition to relational operators. We propose implementing adaptive query execution, so that the engine can change the plan for each query as it sees what data earlier stages produced. This approach was already prototyped in the Shark research project but was never merged into Spark.

Goals

Add adaptive execution to the Spark engine and Spark SQL / DataFrames. This should include:

- An internal API in DAGScheduler to allow submitting individual DAG stages and collecting statistics about their results.
- The ability to change communication operations downstream of a stage, e.g. change the number of tasks for a reduce, or broadcast some data instead of shuffling it.
- Planner changes in Spark SQL that use this to:
  - Set the number of reduce tasks for aggregation.
  - Select join strategies, including changing from shuffle join to broadcast join.

Some non-goals for the first version:

- Public API: it would be good to expose this API stably long-term (for third-party libraries), but for now we should keep it private to gain experience.
- Adaptive planning for core RDD operations: this is more difficult because the RDD API includes a Partitioner and number of partitions. It would require some larger changes to RDD, which can be investigated later. For the same reason, adaptive planning may not be possible for DataFrames converted to RDDs with DataFrame.rdd().

Architecture

The main idea will be to run DAG stages implementing the “map” side of a shuffle first, before deciding how many reduce tasks to use or whether to use another communication pattern, such as broadcast. We will configure the map tasks produce a reasonably large number of output partitions (e.g. 1000), then look at the output sizes of these partitions (plus maybe other stats) and decide how many reduce tasks to use, which partitions to fetch in each one, or whether to do something different such as having all reduce tasks fetch certain partitions (for broadcast). To do this, reduce tasks will have to be able to fetch multiple map output partitions, but there is already groundwork in place for this.

For example, imagine we are joining two RDDs, A and B, computed through map functions. We will first run the map tasks for both A and B, and create output files with 1000 partitions on each node. The maps will also tell the driver the size in bytes of each output partition. Given this, the driver app (e.g. Spark SQL) can decide on any of the following strategies for the join:

- Run a shuffle join, where each reduce task fetches some of the 1000 partitions from both RDDs. The driver can choose the number of tasks and which partitions go to each task based on their sizes, e.g. to deal with skew.
- Broadcast all outputs from A to the nodes that have outputs from B. To do this, we’d launch reduce tasks that each read one mapper’s worth of outputs from B, and all partitions from A.
- Broadcast some partitions from A but not others. This can be done to handle skew, where some partitions have very popular keys (e.g. NULL) and others don’t.

The full implementation will depend on three pieces:

- DAG Scheduler: allow submitting map stages with no downstream reduce stage and collecting output statistics.
- Shuffle: allow reduce tasks to fetch multiple map output partitions.
- Spark SQL: implement the actual adaptive query planning, i.e. submit map stages separately and change downstream operators based on output statistics.

DAG Scheduler

We add a new method, DAGScheduler.submitMapStage(), that takes a ShuffleDependency representing the given map stage, runs the tasks, and returns a MapOutputStatistics object providing the total bytes of output for each partition. This object will be wrapped in a JobWaiter object so that the caller can also cancel the stage.

Once a map stage finishes, or even before, downstream stages can be submitted with the same ShuffleDependency and will automatically fit into DAGScheduler's dependency tracking. These stages may request multiple partitions from the map stage in each task using the shuffle layer changes described next. From the point of view of the DAGScheduler, each map stage is a barrier, so it doesn't matter which partitions you request. All the existing fault tolerance and cache awareness logic in DAGScheduler will also work as before with this design.

Shuffle Layer

SPARK-2044 introduced a more general API for reduce tasks, where they can request a range of output partitions from the mappers. However, the current implementation in HashShuffleFetcher can only fetch one partition at once. We need to make it actually request multiple blocks, which should not be difficult.

In addition to requesting multiple blocks, two optimizations may be useful in this layer:

1. Allow shuffle fetchers to request that some blocks be cached on the receiving node, to better support broadcasts. Note that this can also be done at the "application layer" (e.g. in Spark SQL) however, and might be more efficient to do there (e.g. it would allow us to deserialize the data only once instead of each time a task requests it).
2. Update the block manager's shuffle block server to support fetching multiple consecutive partition IDs in one disk read, as opposed to separate getBlock calls. This would likely require some "virtual block IDs" to represent a range of blocks in a map output file.

Spark SQL

To execute a query adaptively, we can add a new kind of Exchange operator (for now, let's call it AdaptiveExchange) that submits ShuffleMapStages for its child operators if necessary and then creates a ShuffledRDD based on the stats we get. In Spark SQL, there are two ways to trigger Spark jobs:

- Spark jobs are triggered when a user directly invokes an action of a DataFrame, e.g. df.collect() and df.write.save(...).
- Users can first get the RDD representation of a DataFrame (through df.rdd) and then invoke RDD's actions (e.g. df.rdd.map(...).foreach(...)).

Since AdaptiveExchange will eagerly submit stages, we can first apply adaptive query execution when users invoke actions of a DataFrame. When a user tries to get the RDD representation of a DataFrame and wants to apply transformations on it, we will not use adaptive query execution for now.

Runtime Decisions

In the initial implementation, we can target supporting the following three runtime decisions.

Setting the Number of Reducers

First, we can determine the number of reducers at runtime. Taking an aggregate operator as an example, we can initially create a relative large number of reducer partitions (e.g. 1000) and set a target data size per reducer. Then, based on this target size per reducer, we pack those initial reducer partitions and determine what partitions a real reducer should fetch. For example, if we set 64 MB as the target size per reducer and we have 5 initial reducer partitions with size 70 MB, 30MB, 20 MB, 20MB, and 50 MB. We will have 3 real reducers. The first reducer will fetch the first initial reducer partition; the second reduces will fetch the second to fourth initial reducer partitions; and the third reducer will fetch the last initial reducer partition.

Determining the Join Strategy

Second, we can determine the Join strategy used at runtime. For a complex query, a Join operator may take intermediate results as input tables. At runtime, we can figure out the size of input tables of a Join operator. Then, if a input table is small enough, we can use broadcast join instead of shuffle join. This will require a threshold for how small a table should be (ideally the same as the current broadcast join threshold setting).

Handling Skewed Shuffle Join Input Data

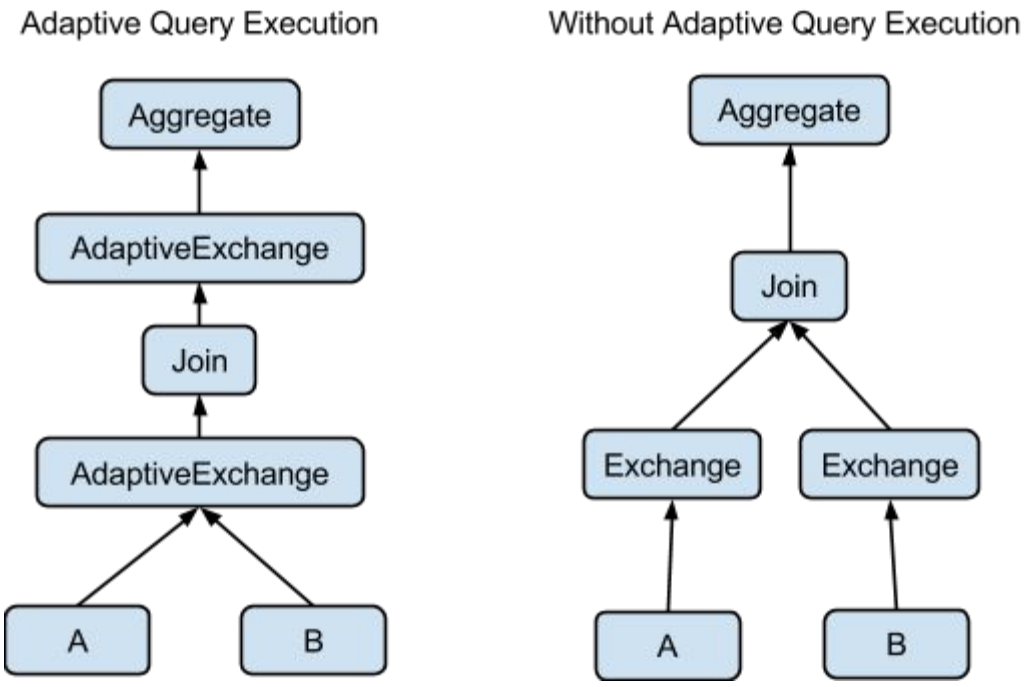
Third, we can use adaptive query execution to handle data skew when joining two tables. For a Join operator, if the size of one or multiple initial reducer partitions of one input table is very large and the size of those corresponding initial reducer partitions of another input table is small, we can use broadcast join to handle these skewed partitions and use shuffle join for other partitions. For example, the size of initial reducer partitions of two input tables of a join operator can be table1: [2000 MB, 50 MB, 60 MB, 70 MB, 100 MB], and table2: [10MB, 20 MB, 50 MB, 40 MB, 50 MB]. For this case, we can broadcast the first partition of table 2. So, we will not shuffle rows of the first partition of table1 to a single reducer.

Required Changes

Query Compilation

At query compilation time, whenever we find a physical operator that defines its requiredChildDistribution field (it means this operator has specific requirement on how its input rows are organized), we add an AdaptiveExchange operator. This logic is different from our current physical planning logic, which uses EnsureRequirements to add Exchange operators and tries to avoid add unnecessary Exchange operators. Unlike an Exchange operator, having an AdaptiveExchange before a physical operator does not mean we will always shuffle data. At runtime, an AdaptiveExchange operator will determine if to shuffle data based on the outputPartitioning of its child and its requiredChildDistribution field. Below, we show an example query and its physical plan tree (the plan generated from our existing planner is also shown).

```
SELECT ...
FROM A JOIN B ON (A.key = B.key)
GROUP BY A.key
```



In the query plan for adaptive query execution, the AdaptiveExchange operator appearing before the Join operator has two children and it can plan how to shuffle the data by considering stats from both table A and table B. The AdaptiveExchange operator appearing before the Aggregate operator will check if its input rows already satisfy the required distribution. If so, it is a NoOp operator.

Runtime Execution

At runtime, AdaptiveExchange's's doExecute method is used to drive the query execution. When AdaptiveExchange's doExecute is called, it will first get RDDs of its children operators and submit ShuffleMapStages for its children. Then, it is blocked until those stages get finished. Once those submitted stages get finished, the AdaptiveExchange collects stats and calls its planning strategies to determine how to create the ShuffledRDD used to fetch map output (this ShuffledRDD is the first RDD at the reducer side). Finally, the AdaptiveExchange operator updates its outputPartitioning field (e.g. setting the number of partitions it actually uses) and the doExecute method returns the created ShuffledRDD.

Tasks

There are four tasks for the MVP of this work on the Spark SQL side.

1. Implement AdaptiveExchange operator and hook it in our query planner with a feature flag.
2. For different operators, add planning strategies to the AdaptiveExchange operator. We can first start with determining number of reducers and Join plan (broadcast join or shuffled join).
3. If necessary, implement new operators that work with AdaptiveExchange. For example, we may want to write a hybrid inner equi join operator that takes an AdaptiveExchange as input.
4. Set the default values of parameters that affect adaptive planning (e.g. amount of data per reduce task) based on a benchmark such as TPC-DS.

Limitations and Extensions

The main limitation of the proposed approach is that we have to use a fairly large number of map output partitions, which may create overhead in the shuffle writer and reader. This should be much less if we add a binary interface to the shuffle layer in the future, e.g. as part of Tungsten (where a caller could just directly write a sorted binary file and provide the partition offsets within that). For now, the # of partitions can be set based on the size of the input data set, for example, so that it's smaller if the input is small.

Other future extensions could include the following:

- Using this in libraries beyond Spark SQL: streaming may be a good candidate, though in streaming we can also just change the plan on the next batch of data.
- Applying this to Spark Core: this would require changing the RDD interface slightly to allow computing a RDD's Partitioner and partitions after its parent stages have run. It may not be too hard to do it since both of these are accessed by getter methods, but

we'd likely need a special type of Dependency that tells the DAGScheduler to compute the parents first before probing this RDD, or a getPartitions / getPartitioner that can optionally take a set of MapOutputStatistics for the parent stages.

- Using more statistics than partition output sizes: we can easily collect these statistics using accumulators while each map stage is running.
- More types of joins in SQL, e.g. different per-partition strategies to deal with skew.