Hive on Spark

Xuefu Zhang, Reynold Xin
06-25-2014

1. Introduction

We propose modifying Hive to add Spark as a third execution backend, parallel to MapReduce and Tez.

Spark is an open-source data analytics cluster computing framework that's built outside of Hadoop's two-stage MapReduce paradigm but on top of HDFS. Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop `InputFormat`s (such as HDFS files) or by transforming other RDDs. By being applied by a series of transformations such as `groupBy` and `filter`, or actions such as `count` and `save` that are provided by Spark, RDDs can be processed and analyzed to fulfill what MapReduce jobs can do without having intermediate stages.

SQL queries can be easily translated into Spark transformation and actions, as demonstrated in Shark and Spark SQL. In fact, many primitive transformations and actions are SQL-oriented such as `join` and `count`.

More information about Spark can be found here:

- Apache Apark page: http://spark.apache.org/
- Apache Spark blogpost: http://blog.cloudera.com/blog/2013/11/putting-spark-to-use-fast-in-memory-computing-for-your-big-data-applications/
- Apache Spark JavaDoc: http://spark.apache.org/docs/1.0.0/api/java/index.html

1.1 Motivation

Here are the main motivations for enabling Hive to run on Spark:

1. Spark user benefits: This feature is very valuable to users who are already using Spark for other data processing and machine learning needs. Standardizing on one execution backend is convenient for operational management, and makes it easier to develop expertise to debug issues and make enhancements.
2. Greater Hive adoption: Following the previous point, this brings Hive into the Spark user base as a SQL on Hadoop option, further increasing Hive's adoption.
3. Performance: Hive queries, especially those involving multiple reducer stages, will run faster, thus improving user experience as Tez does.

It is not a goal for the Spark execution backend to replace Tez or MapReduce. It is healthy for the Hive project for multiple backends to coexist. Users have a choice whether to use Tez, Spark or

MapReduce. Each has different strengths depending on the use case. And the success of Hive does not completely depend on the success of either Tez or Spark.

1.2 Design Principle

The main design principle is to have no or limited impact on Hive's existing code path and thus no functional or performance impact. That is, users choosing to run Hive on either MapReduce or Tez will have existing functionality and code paths as they do today. In addition, plugging in Spark at the execution layer keeps code sharing at maximum and contains the maintenance cost, so Hive community does not need to make specialized investments for Spark.

Meanwhile, users opting for Spark as the execution engine will automatically have all the rich functional features that Hive provides. Future features (such as new data types, UDFs, logical optimization, etc) added to Hive should be automatically available to those users without any customization work to be done done in Hive's Spark execution engine.

1.3 Comparison with Shark and Spark SQL

There are two related projects in the Spark ecosystem that provide Hive QL support on Spark: Shark and Spark SQL.

- The Shark project translates query plans generated by Hive into its own representation and executes them over Spark.
- Spark SQL is a feature in Spark. It uses Hive's parser as the frontend to provide Hive QL support. Spark application developers can easily express their data processing logic in SQL, as well as the other Spark operators, in their code. Spark SQL supports a different use case than Hive.

Compared with Shark and Spark SQL, our approach by design supports all existing Hive features, including Hive QL (and any future extension), and Hive's integration with authorization, monitoring, auditing, and other operational tools.

1.4 Other Considerations

We know that a new execution backend is a major undertaking. It inevitably adds complexity and maintenance cost, even though the design avoids touching the existing code paths. And Hive will now have unit tests running against MapReduce, Tez, and Spark. We think that the benefit outweighs the cost. From an infrastructure point of view, we can get sponsorship for more hardware to do continuous integration.

Lastly, Hive on Tez has laid some important groundwork that will be very helpful to support a new execution engine such as Spark. This project here will certainly benefit from that. On the other hand, Spark is a framework that's very different from either MapReduce or Tez. Thus, it's very

likely to find gaps and hiccups during the integration. It's expected that Hive community will work closely with Spark community to ensure the success of the integration.

2. High-Level Functionality

2.1 A New Execution Engine

We will introduce a new execution, Spark, in addition to existing MapReduce and Tez. To use Spark as an execution engine in Hive, set the following:

```
set hive.execution.engine=spark;
```

The default value for this configuration is still "`mr`". Hive continues to work on MapReduce and Tez as is on clusters that don't have spark.

The new execution engine should support all Hive queries without requiring any modification of the queries. Query result should be functionally equivalent to that from either MapReduce or Tez.

2.2 Spark Configuration

When Spark is configured as Hive's execution, a few configuration variables will be introduced such as the master URL of the Spark cluster. However, they can be completely ignored if Spark isn't configured as the execution engine.

2.3 Miscellaneous Functionality

1. Hive will display a task execution plan that's similar to that being displayed in "`explain`" command for MapReduce and Tez.
2. Hive will give appropriate feedback to the user about progress and completion status of the query when running queries on Spark.
3. The user will be able to get statistics and diagnostic information as before (counters, logs, and debug info on the console).

3. Hive-Level Design

As noted in the introduction, this project takes a different approach from that of Shark or Spark SQL in the sense that we are not going to implement SQL semantics using Spark's primitives. On the contrary, we will implement it using MapReduce primitives. The only new thing here is

that these MapReduce primitives will be executed in Spark. In fact, only a few of Spark's primitives will be used in this design.

The approach of executing Hive's MapReduce primitives on Spark that is different from what Shark or Spark SQL does has the following direct advantages:

1. Spark users will automatically get the whole set of Hive's rich features, including any new features that Hive might introduce in the future.
2. This approach avoids or reduces the necessity of any customization work in Hive's Spark execution engine.
3. It will also limit the scope of the project and reduce long-term maintenance by keeping Hive-on-Spark congruent to Hive MapReduce and Tez.

The main work to implement the Spark execution engine for Hive lies in two folds: query planning, where Hive operator plan from semantic analyzer is further translated a task plan that Spark can execute, and query execution, where the generated Spark plan gets actually executed in the Spark cluster. Of course, there are other functional pieces, miscellaneous yet indispensable such as monitoring, counters, statistics, etc. Some important design details are thus also outlined below.

It's worth noting that though Spark is written largely in Scala, it provides client APIs in several languages including Java. Naturally we choose Spark Java APIs for the integration, and no Scala knowledge is needed for this project.

3.1 Query Planning

Currently for a given user query Hive semantic analyzer generates an operator plan that's composed of a graph of logical operators such as `TableScanOperator`, `ReduceSink`, `FileSink`, `GroupByOperator`, etc. `MapReduceCompiler` compiles a graph of `MapReduceTask`s and other helper tasks (such as `MoveTask`) from the logical, operator plan. Tez behaves similarly, yet generates a `TezTask` that combines otherwise multiple MapReduce tasks into a single Tez task.

For Spark, we will introduce `SparkCompiler`, parallel to `MapReduceCompiler` and `TezCompiler`. Its main responsibility is to compile from Hive logical operator plan a plan that can be execute on Spark. Thus, we will have `SparkTask`, depicting a job that will be executed in a Spark cluster, and `SparkWork`, describing the plan of a Spark task. Thus, `SparkCompiler` translates a Hive's operator plan into a `SparkWork` instance.

During the task plan generation, `SparkCompiler` may perform physical optimizations that's suitable for Spark. However, for first phase of the implementation, we will focus less on this unless it's easy and obvious. Further optimization can be done down the road in an incremental manner as we gain more and more knowledge and experience with Spark.

How to generate `SparkWork` from Hive's operator plan is left to the implementation. However, there seems to be a lot of common logics between Tez and Spark as well as between MapReduce and Spark. If feasible, we will extract the common logic and package it into a shareable form, leaving the specific implementations to each task compiler, without destabilizing either MapReduce or Tez.

3.2 Job Execution

A `SparkTask` instance can be executed by Hive's task execution framework in the same way as for other tasks. Internally, the `SparkTask.execute()` method will make RDDs and functions out of a `SparkWork` instance, and submit the execution to the Spark cluster via a Spark client.

Once the Spark work is submitted to the Spark cluster, Spark client will continue to monitor the job execution and report progress. A Spark job can be monitored via `SparkListener` APIs. Currently not available in Spark Java API, We expect they will be made available soon with the help from Spark community.

With `SparkListener` APIs, we will add a `SparkJobMonitor` class that handles printing of status as well as reporting the final result. This class provides similar functions as `HadoopJobExecHelper` used for MapReduce processing, or `TezJobMonitor` used for Tez job processing, and will also retrieve and print the top level exception thrown at execution time, in case of job failure.

Spark job submission is done via a `SparkContext` object that's instantiated with user's configuration. When a `SparkTask` is executed by Hive, such context object is created in the current user session. With the context object, RDDs corresponding to Hive tables are created and `MapFunction` and `ReduceFunction` (more details below) that are built from Hive's `SparkWork` and applied to the RDDs. Job execution is triggered by applying a `foreach()` transformation on the RDDs with a dummy function.

One `SparkContext` per user session is right thing to do, but it seems that Spark assumes one `SparkContext` per application because of some thread-safety issues. We expect that Spark community will be able to address this issue timely.

3.3 Design Considerations

This section covers the main design considerations for a number of important components, either new that will be introduced or existing that deserves special treatment. For other existing components that aren't named out, such as UDFs and custom Serdes, we expect that special considerations are either not needed or insignificant.

Table as RDD

A Hive table is nothing but a bunch of files and folders on HDFS. Spark primitives are applied to RDDs. Thus, naturally Hive tables will be treated as RDDs in the Spark execution engine. However, Hive table is more complex than a HDFS file. It can have partitions and buckets, dealing with heterogeneous input formats and schema evolution. As a result, the treatment may not be that simple, potentially having complications, which we need to be aware of.

It's possible we need to extend Spark's Hadoop RDD and implement a Hive-specific RDD. While RDD extension seems easy in Scala, this can be challenging as Spark's Java APIs lack such capability. We will find out if RDD extension is needed and if so we will need help from Spark community on the Java APIs.

SparkWork

As discussed above, `SparkTask` will use `SparkWork`, which describes the task plan that the Spark job is going to execute upon. `SparkWork` will be very similar to `TezWork`, which is basically composed of `MapWork` at the leaves and `ReduceWork` (occassionally, `UnionWork`) in all other nodes.

Defining `SparkWork` in terms of `MapWork` and `ReduceWork` makes the new concept easier to be understood. The "`explain`" command will show a pattern that Hive users are familiar with.

SparkTask

To execute the work described by a `SparkWork` instance, some further translation is necessary, as `MapWork` and `ReduceWork` are MapReduce-oriented concepts, and implementing them with Spark requires some traverse of the plan and generation of Spark constructs (RDDs, functions). How to traverse and translate the plan is left to the implementation, but this is very Spark specific, thus having no exposure to or impact on other components.

Above mentioned `MapFunction` will be made from `MapWork`, specifically, the operator chain starting from `ExecMapper.map()` method. `ExecMapper` class implements MapReduce Mapper interface, but the implementation in Hive contains some code that can be reused for Spark. Therefore, we will likely extract the common code into a separate class, `MapperDriver`, to be shared by both MapReduce and Spark. Note that this is just a matter of refactoring rather than redesigning.

(Tez probably had the same situation. However, Tez has chosen to create a separate class, `RecordProcessor`, to do something similar.)

Similarly, `ReduceFunction` will be made of `ReduceWork` instance from `SparkWork`. To Spark, `ReduceFunction` has no difference from `MapFunction`, but the function's

implementation will be different, made of the operator chain starting from `ExecReducer.reduce()`. Also because some code in `ExecReducer` are to be reused, likely we will extract the common code into a separate class, `ReducerDriver`, so as to be shared by both MapReduce and Spark.

All functions, including `MapFunction` and `ReduceFunction` needs to be serializable as Spark needs to ship them to the cluster. This could be tricky as how to package the functions impacts the serialization of the functions, and Spark is implicit on this.

Note that Spark's built-in map and reduce transformation operators are functional with respect to each record. For example, Hive's operators, however, need to be initialized before being called to process rows and be closed when done processing. `MapFunction` and `ReduceFunction` will have to perform all those in a single `call()` method. For the purpose of using Spark as an alternate execution backend for Hive, we will be using the `mapPartitions` transformation operator on RDDs, which provides an iterator on a whole partition of data. With the iterator in control, Hive can initialize the operator chain before processing the first row, and de-initialize it after all input is consumed.

It's worth noting that during the prototyping Spark caches function globally in certain cases, thus keeping stale state of the function. Such culprit is hard to detect and hopefully Spark will be more specific in documenting features down the road.

Shuffle, Group, and Sort

While this comes for “free” for MapReduce and Tez, we will need to provide an equivalent for Spark. Fortunately, Spark provides a few transformations that are suitable to substitute MapReduce’s shuffle capability, such as `partitionBy`, `groupByKey`, and `sortByKey`. Transformation `partitionBy` does pure shuffling (no grouping or sorting), `groupByKey` does shuffling and grouping, and `sortByKey()` does shuffling plus sorting. Therefore, for each `ReduceSinkOperator` in `SparkWork`, we will need to inject one of the transformations.

Having the capability of selectively choosing the exact shuffling behavior provides opportunities for optimization. For instance, Hive's `groupBy` doesn't require the key to be sorted, but MapReduce does it nevertheless. In Spark, we can choose `sortByKey` only if necessary key order is important (such as for SQL `order by`).

While `sortByKey` provides no grouping, it’s easy to group the keys as rows with the same key will come consecutively. On the other , `groupByKey` clusters the keys in a collection, which naturally fits the MapReduce’s reducer interface.

As Hive is more sophisticated in using MapReduce keys to implement operations that’s not directly available such as `join`, above mentioned transformations may not behave exactly as Hive needs. Thus, we need to be diligent in identifying potential issues as we move forward.

Finally, it seems that Spark community is in the process of improving/changing the shuffle related APIs. Thus, this part of design is subject to change. Please refer to https://issues.apache.org/jira/browse/SPARK-2044 for the details on Spark shuffle-related improvement.

Join

It's rather complicated in implementing `join` in MapReduce world, as manifested in Hive. Hive has reduce-side `join` as well as map-side `join` (including map-side hash lookup and map-side sorted merge). We will keep Hive's `join` implementations. However, extra attention needs to be paid on the shuffle behavior (key generation, partitioning, sorting, etc), since Hive extensively uses MapReduce's shuffling in implementing reduce-side `join`. It's expected that Spark is, or will be, able to provide flexible control over the shuffling, as pointed out in the previous section(Shuffle, Group, and Sort).

Number of Tasks

As specified above, Spark transformations such as `partitionBy` will be used to connect mapper-side's operations to reducer-side's operations. The number of partitions can be optionally given for those transformations, which basically dictates the number of reducers.

The determination of the number of reducers will be the same as it's for MapReduce and Tez.

Local MapReduce Tasks

While we could see the benefits of running local jobs on Spark, such as avoiding sinking data to a file and then reading it from the file to memory, in the short term, those tasks will still be executed the same way as it is today. This means that Hive will always have to submit MapReduce jobs when executing locally. However, this can be further investigated and evaluated down the road.

The same applies for presenting the query result to the user. Presently, a fetch operator is used on the client side to fetch rows from the temporary file (produced by `FileSink` in the query plan). It's possible to have the `FileSink` to generate an in-memory RDD instead and the fetch operator can directly read rows from the RDD. Again this can be investigated and implemented as a future work.

Semantic Analysis and Logical Optimizations

Neither semantic analyzer nor any logical optimizations will change. Physical optimizations and MapReduce plan generation have already been moved out to separate classes as part of Hive on Tez work.

Job Diagnostics

Basic “job succeeded/failed” as well as progress will be as discussed in “Job monitoring”. Hive’s current way of trying to fetch additional information about failed jobs may not be available immediately, but this is another area that needs more research.

Spark provides WebUI for each `SparkContext` while it’s running. Note that this information is only available for the duration of the application by default. To view the web UI after the fact, set `spark.eventLog.enabled` to `true` before starting the application. This configures Spark to log Spark events that encode the information displayed in the UI to persisted storage.

Spark’s Standalone Mode cluster manager also has its own web UI. If an application has logged events over the course of its lifetime, then the Standalone master’s web UI will automatically re-render the application’s UI after the application has finished.

If Spark is run on Mesos or YARN, it is still possible to reconstruct the UI of a finished application through Spark’s history server, provided that the application’s event logs exist.

For more information about Spark monitoring, visit http://spark.apache.org/docs/latest/monitoring.html.

Counters and Metrics

Spark has accumulators which are variables that are only “added” to through an associative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can add support for new types. In Hive, we may use Spark accumulators to implement Hadoop counters, but this may not be done right way.

Spark publishes runtime metrics for a running job. However, it’s very likely that the metrics are different from either MapReduce or Tez, not to mention the way to extract the metrics. The topic around this deserves a separate document, but this can be certainly improved upon incrementally.

Explain Statements

Explain statements will be similar to that of `TezWork`.

Hive Variables

Hive variables will continue to work as it is today. The variables will be passed through to the execution engine as before. However, some execution engine related variables may not be applicable to Spark, in which case, they will be simply ignored.

Union

While it's mentioned above that we will use MapReduce primitives to implement SQL semantics in the Spark execution engine, union is one exception. While it's possible to implement it with MapReduce primitives, it takes up to three MapReduce jobs to union two datasets. Using Spark's union transformation should significantly reduce the execution time and promote interactivity.

In fact, Tez has already deviated from MapReduce practice with respect to union. There is an existing `UnionWork` where a union operator is translated to a work unit.

Concurrency and Thread Safety

Spark launches mappers and reducers differently from MapReduce in that a worker may process multiple HDFS splits in a single JVM. However, Hive's map-side operator tree or reduce-side operator tree operates in a single thread in an exclusive JVM. Reusing the operator trees and putting them in a shared JVM with each other will more than likely cause concurrency and thread safety issues. Such problems, such as static variables, have surfaced in the initial prototyping. For instance, variable `ExecMapper.done` is used to determine if a mapper has finished its work. If two `ExecMapper` instances exist in a single JVM, then one mapper that finishes earlier will prematurely terminate the other also. We expect there will be a fair amount of work to make these operator tree thread-safe and contention-free. However, this work should not have any impact on other execution engines.

Build Infrastructure

There will be a new "ql" dependency on Spark. Currently Spark client library comes in a single jar. The spark jar will be handled the same way Hadoop jars are handled: they will be used during compile, but not included in the final distribution. Rather we will depend on them being installed separately. The spark jar will only have to be present to run Spark jobs, they are not needed for either MapReduce or Tez execution.

On the other hand, to run Hive code on Spark, certain Hive libraries and their dependencies need to be distributed to Spark cluster by calling `SparkContext.addJar()` method. As Spark also depends on Hadoop and other libraries, which might be present in Hive's dependents yet with different versions, there might be some challenges in identifying and resolving library conflicts. Jetty libraries posted such a challenge during the prototyping.

Mini Spark Cluster

Spark jobs can be run local by giving "`local`" as the master URL. Most testing will be performed in this mode. In the same time, Spark offers a way to run jobs in a local cluster, a cluster made of a given number of processes in the local machine. We will further determine if this is a good way to run Hive's Spark-related tests.

Testing

Testing, including pre-commit testing, is the same as for Tez. Currently Hive has a coverage problem as there are a few variables that requires full regression suite run, such as Tez vs MapReduce, vectorization on vs off, etc. We propose rotating those variables in pre-commit test run so that enough coverage is in place while testing time isn't prolonged.

3.4 Potentially Required Work from Spark

During the course of prototyping and design, a few issues on Spark have been identified, as shown throughout the document. Potentially more, but the following is a summary of improvement that's needed from Spark community for the project:

1. Job monitoring API in Java.
2. `SparkContext` thread safety issue.
3. Improve shuffle functionality and API.
4. Potentially, Java API for extending RDD.

4. Summary

It can be seen from above analysis that the project of Spark on Hive is simple and clean in terms of functionality and design, while complicated and involved in implementation, which may take significant time and resources. Therefore, we are going to take a phased approach and expect that the work on optimization and improvement will be on-going in a relatively long period of time while all basic functionality will be there in the first phase.

Secondly, we expect the integration between Hive and Spark will not be always smooth. Functional gaps may be identified and problems may arise. We anticipate that Hive community and Spark community will work closely to resolve any obstacles that might come on the way.

Nevertheless, we believe that the impact on existing code path is minimal. While Spark execution engine may take some time to stabilize, MapReduce and Tez should continue working as it is.

5. Acknowledgement

This document has received valuable and constructive feedback from both Apache Hive community and Apache Spark community. My special thanks go to Carl Steinbach and Yin Huai as well as to Matei Zaharia and Patrick Wendell for their input and advice in the process of research and design.