FP7 Project ASAP

Adaptable Scalable Analytics Platform

ASAP D4.1 Execution Engine Design

WP 4 – Dependency-aware query execution engine

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	15 Feb 2015	P. Pratikakis, S. Papagiannaki, M. Chalkiadaki	Initial Version
0.2	27 Feb 2015	S. Papagiannaki, M. Chalkiadaki	First Revision
1.0	5 Mar 2015	P. Pratikakis	Final Version

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Executive Summary

This document presents the current design of the execution engine for recursive analytics queries, as developed in WP4 of project ASAP. The execution engine design is an extension of the Spark analytics engine. We extend the Spark scheduling algorithm to allow for ongoing analytics queries to issue sub-queries recursively, by modifying the scheduling actors of Spark to forward query initializiation and completion messages to the scheduler node. We avoid centralizing the scheduling algorithm by optimizing for direct communication between worker nodes whenever possible, to avoid congesion at the scheduler node.

Contents

```
val file1 = sc.textFile("hdfs://file1")
val file2 = sc.textFile("hdfs://file2")
file1.map(word1 =>
  file2.filter(word2 =>
    (word1.length > word2.length))
      .collect())
  .collect()
```

Figure 1: example of nested RDD operations

1 Introduction

The main objective of this Work Package is the design and an development of a dependency-aware query execution engine which incorporates the following functionalities:

- the division of query computations into computation tasks and the representation of them in the system;
- the analysis of tasks to discover data dependencies;
- the data placement constraints posed by each data store and data schema, and their representation in the runtime system;
- the scheduler of computation tasks to computation nodes, while taking into account the data location and data dependencies.

1.1 Task Description

The Task T4.1, which aims at producing Deliverable D4.1, describes the detailed design and an early implementation of the dependence analysis, the scheduler and the execution engine.

2 Dependency-aware query execution engine

As a base for the dependency-aware query execution engine we employed the Spark [1] execution engine. Spark uses an abstraction for describing general purpose calculations on datasets by keeping track of lineage dependencies between the required dataset transformations. Moreover, it contains a scheduling mechanism for decomposing the calculations in pipelined tasks that can be executed independently in a cluster by taking into account locality and resource constraints. Our design extends the Spark execution in order to enable the execution of nested calculations like the one in Figure 1.

2.1 Dependence analysis

The fundamental abstraction in Spark are RDDs (Resilient Distributed Dataset) which are immutable partitioned collections, stored in an external storage system, such as a file in HDFS, or derived by applying operators to other RDDs.

RDDs support two types of operations: transformations which create a new dataset from an existing one, and actions which return a value to the driver program after running a computation on the dataset.

All transformations are lazy, therefore each RDD keeps track of all the transformations applied to the base dataset and they are only materialized when an action requires a result to be returned to the driver program.

Once an action on a RDD is triggered on the driver side, a job is submitted to the scheduler. Each job is decomposed in smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce). The decomposition into stages is achieved by classifying RDD dependencies into narrow and wide. In case of a narrow dependency, each partition of the child RDD is derived by at most one partition of the parent RDD. In case of a wide dependency, each partition of the child RDD is derived by several parent partitions. Hence, each stage contains as many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wide dependencies (or any already computed partitions).

Moreover, the RDD abstraction also enables the data analyst to provide hints how the data should be partitioned and calculated by providing

- partitioners that define how the elements in a key-value pair RDD are partitioned by key and
- a list of preferred locations to compute each partition on (e.g. block locations for an HDFS file)

2.2 Scheduler

The Spark scheduler first examines the RDDs lineage graph to build a DAG of stages. Then, it will try to submit the final stage. However, if the parent stages are not yet available it will recursively force them to be calculated. Whenever a stage's parents are available, the scheduler will launch the necessary tasks in order to compute the missing partitions.

The task scheduler running in the driver side decides which tasks should run in which node based on resource and locality constraints. For instance, if a task needs to process a partition that is available in memory on a node, it will be sent to that node. Otherwise, if a task processes a partition for which the containing RDD provides preferred locations, it will be send it to those locations.

Finally, the SchedulerBackend module, which resides also in the driver program, generates a message containing the serialized task for each task and sends it to the scheduled executor.

2.3 Execution Engine

The executor once receives the task, deserializes it and runs it. Tasks are divided into ResultTasks and ShuffleMapTasks. The final stage consists of various ResultTasks while the intermediate stages consists of ShuffleMapTasks. The output of ResultTasks is sent back to the driver while the output data of the ShuffleMapTasks are written to the local file system waiting for subsequent tasks (reducers) to download them.

Whenever a task requires intermediate data from parent stages will make remote pull requests to download them.

Finally, upon the end of the execution, the executor notifies the driver program about the task execution result status.

2.4 Implementation Details

The Spark engine is implemented in Scala, a functional, object oriented language that is compiled to JVM bytecode.

The Scala concurrency model relies on the Akka library, which implements the actor model. Each Akka actor is a lightweight task that can send or receive messages.

The overview of the scheduling mechanism is depicted in Figure 2. Each bubble represents an Akka actor. The main cluster messages for Spark scheduler-executor communication are:

1. RegisterExecutor : When an executor is initiated, it sends a message to master to register itself
2. LaunchTask : Master sends a serialized task
3. StatusUpdate : The executor updates master with the task state(RUNNING,FAILED,FINISHED)
4. KillTask : Master orders an executor to stop executing a task

However, the Spark would fail to execute the nested calculation in Figure 1. The reason is that some RDD metadata are known only by the driver program while such a calculation requires such an information to be shared also with the executors.

An execution attempt would be the following: The outer collect method forces the computation in the driver program to start. Since no shuffle operations are involved, the DAG graph will consist of only one stage. This stage will contain one transformation of the RDD representing the file1 in the Hadoop to an RDD derived by appliying the *map* function. The scheduler will try to submit this stage and since there are not waiting parent stages it will proceed with creating and submitting the missing tasks. Then the TaskScheduler will create tasks which literally will force the nested code to be executed for each word of the file1. Each task will be serialized and sent to an idle executor. As soon as the executor will receive the task, it will try to apply the computation on its partitions of the RDD. At this point the computation in the spark engine would fail since the executor is missing information in order to perform the computation.

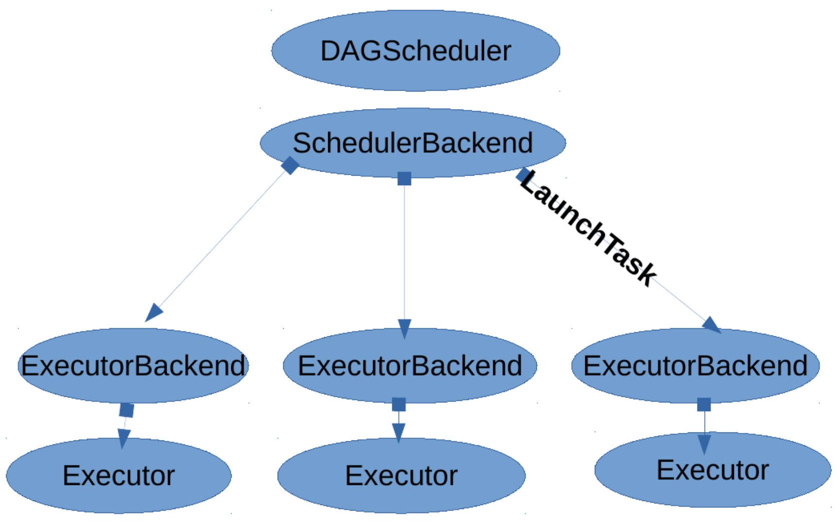

Figure 2: Spark runtime design overview

Therefore, we introduce some extra control messages to the Scheduler-Executor protocol. When the executor tries to invoke the nested map operation, it figures out that it is on executor mode, thus cannot create RDDs, so it sends a CreateRDD message to SchedulerBackend with (rd-did,"map",function) as arguments. Then the scheduler, looks up the RDD with the specified id, and using reflection, invokes the "map" method, creating the desired RDD. Then the Scheduler sends back to the executor the id of the created RDD (SendRDD msg). Now the worker creates promise of the RDD based on the id received. When the nested collect is called the executor sends the CollectRDD message, asking the Scheduler to collect the file2 RDD, and send back the result.

Figure 3 shows the sequence of messages that have to be sent.

3 Benchmarks

To test our early prototype of dependence analysis and scheduler extensions for recursively nested queries, we have used the current implementation of the Peak Detection application, as presented in the Telecommunication Analytics application deliverable D9.2. We have re-implemented the application twice to run on Spark execution engine and also to use our extension of Spark using nested queries. We have run both applications on data sets of various sizes using two clusters of two and five nodes, respectively. Table **??** presents the results of running the original Peak Detection on a single node using SQLite, the "flat" distributed implementation using Spark, and the "nested" distributed implementation.

Note that the nested implementation is the slowest of the three; that is to be expected as it is an early prototype execution engine running a benchmark not designed for it nor requiring nesting

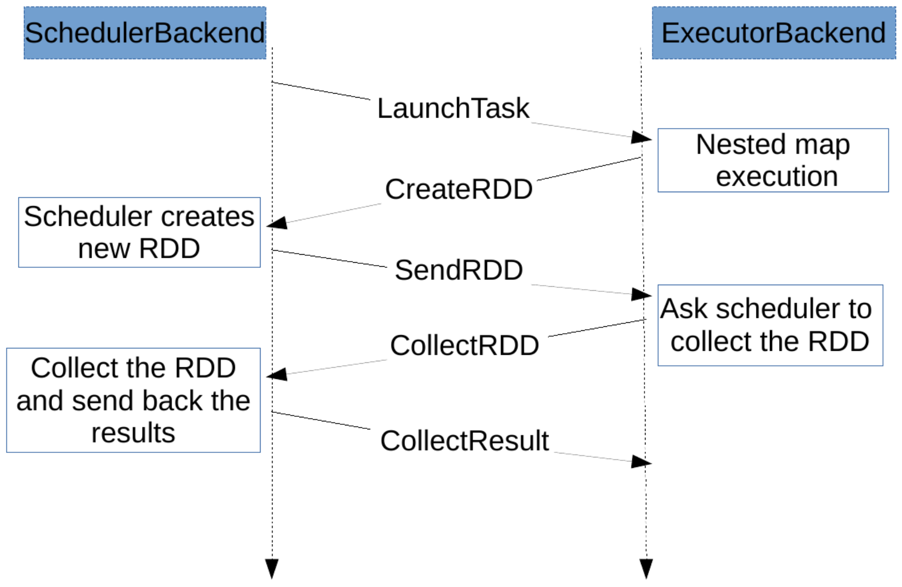

Figure 3: Executor asks the master to perform an RDD operation

Data Size	Original SQLite	Spark 5 nodes	Spark Nested 5 nodes	Spark 2 nodes	Spark Nested 2 nodes
1.2k	0	16	16	11	11
12k	0	16	15	11	11
108k	0	16	16	11	12
1.1M	0	19	21	13	15
11M	1	22	80	15	144
107M	10	35	4120	23	9169

to express. It is always the case that if a query can be expressed as a “flat” computation then that is the best way to schedule it. However, the results satisfy project Milestone MS6, since this early implementation of the nested scheduler satisfies dependencies and runs the application successfuly on both clusters, without any bottlenecks of scalability.

References

[1] Apache Incubator. Spark: Lightning-fast cluster computing, 2013.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D4.1
Execution Engine Design

WP 4 – Dependency-aware query execution engine

Nature: Report

Dissemination: Public