FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D4.2
Execution Engine v.1

WP 4 – Dependency-aware query execution engine

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	20 Feb 2015	P. Pratikakis S. Papagiannaki, P. Katsogridakis	Initial Version
0.2	24 Feb 2015	P. Pratikakis, P. Katsogridakis	First Revision
1.0			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 and implementation 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. Moreover, we add supporting primitives for the recursive, hierarchical decomposition of data using parallel (not iterative) analytics queries, and present the design and early implementation of a distributed scheduler implementation that can parallelize scheduling overheads to allow for finer-grain computations or scale to larger numbers of worker nodes. This deliverable extends D4.1 with additional sections regarding the design and implementations of support for hierarchical data decomposition and distributed scheduling in Spark. Minor changes have also been included in the engine specification section, reflecting any additional information or design decisions taken during the implementation effort in the second year of the project.

Contents

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

Tasks T4.2 and T4.3, which aim at producing Deliverable D4.2, describe the detailed design, implementation, and early testing results of the dependence analysis, the distributed scheduler and the execution engine, as well as the language primitives developed to express hierarchical and recursive computations.

The remainder of this deliverable is organized as follows: The following section repeats and extends the corresponding section in deliverable D4.1 to reflect the current implementation of the nesting query extensions to the Spark scheduler; the subsequent two sections describe the design and implementation of two additional Spark extensions, namely primitives for recursive/hierarchical decomposition of data, as well as the algorithm for distributing the scheduling load to more than one schedulers.

2 Dependency-aware query execution engine

As a base for the dependency-aware query execution engine we employed the Spark [2] 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.

```
1 val file1 = sc.textFile("hdfs://file1")
2 val file2 = sc.textFile("hdfs://file2")
3 file1.map(word1 =>
4   file2.filter(word2 =>
5     (word1.length > word2.length))
6       .collect())
7   .collect()
```

Figure 1: Example of nested RDD operations

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)

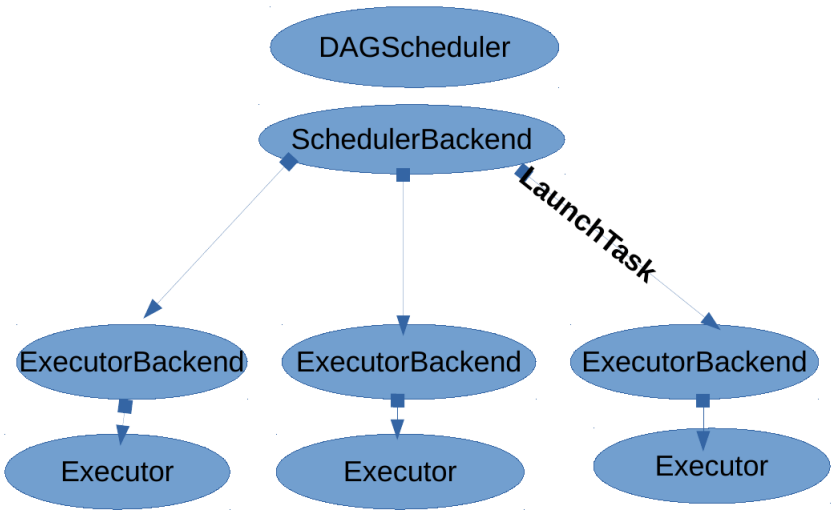

Figure 2: Spark runtime design overview

4. KillTask : Master orders an executor to stop executing a task

However, 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.

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 (rddid,"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.

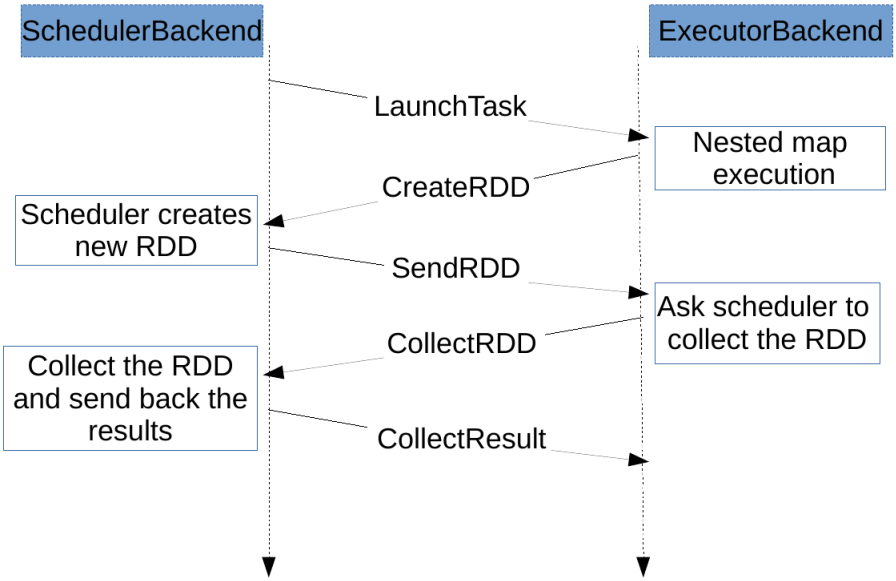

Figure 3: Executor asks the master to perform an RDD operation

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

2.5 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 1 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 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.

Data Size	Original SQLite	Spark 5 nodes	Spark Nested 5 nodes	Spark 2 nodes	Spark Nested 2 nodes
(bytes)	(sec)	(sec)	(sec)	(sec)	(sec)
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

Table 1: Performance results

3 Queries with hierarchical data decomposition

This section describes an extension of the Spark analytics engine that facilitates the hierarchical decomposition of data. A representative example of hierarchical decomposition computation is clustering, particularly hierarchical classification. Clustering has become an essential application for modern data analytics, with K-means being the algorithm most commonly used. Hierarchical clustering offers a tree-structured representation of data, useful for building a hierarchy of clusters. However implementing a scalable version of the hierarchical K-means can be very challenging. Apache Spark is the most commonly used open source engine for big data processing originating from the Map-Reduce programming model, that provides a user friendly mechanism for data transformations, called RDDs. We present an extension of the Spark RDD mechanism in order to express hierarchical structures. We implemented and measured the bisecting KMeans algorithm this way in a Spark cluster, and show preliminary results with up to 40% performance gain with respect to the vanilla implementation.

3.1 Introduction

K-means [4] is the most common algorithm used in cluster analysis, which aims to partition the elememts into k clusters, such that each elemenent belongs to the nearest cluster. K-means is an approximation algorithm, that iterates through the data till the error is minimized. In each iteration, first all elements are assinged to the closest centroid using Euclidean or some other distance metric, and then for each cluster the new centroid is calculated.

Hierarchical K-means is an interesting variation of K-means clustering, that builds a dendrogram on the clustered data, often used in bioinformatics, document clustering and machine learning. The most common algorithms for hierarchical clustering are bisecting K-means, and agglomerative clustering [3]. We focus on the bisecting variation, that splits the elements in a top-down way. The basic steps are:

1. Select a cluster to split
2. Split the selected cluster into 2 sub-clusters using default K-means
3. Repeat step 2 for some iterations and select the best (minimizing error) clusters
4. Repeat the above steps until the requested number of clusters or granularity has been reached

In short, the algorithm uses the output of classification to split data and recursively classify and split these sets, down to a threshold size.

Describing hierarchical clustering in massive datasets is challenging, as one necessarily describes the computation as iterative analytics queries. We propose a high level abstraction to express hierarchical structures. Specifically, we present a higher level way of expressing hierarchical algorithms (while still using the MapReduce abstraction), that can assist the execution engine to more efficiently schedule such computations.

MapReduce [1] is the most popular programming model for large scale cluster computing. A MapReduce cluster is organized in a master-slave architecture. The master is responsible for maintaining the task metadata and scheduling data parallel tasks, using the locality of the data to help with scheduling and load balancing. The worker nodes simply execute the map reduce steps. The MapReduce execution consists of two phases. At the map phase the workers process all the elements of a dataset partition and emit tuples of (key,value). Then, they sort those tuples based on the keys and distribute them over the network, as input to the reduce phase. At the reduce phase the worker takes as input a key and a list of values and generates the final result.

Apache Spark [6] is an extension of the MapReduce programming model that provides the programmer with a rich set of operations on immutable data, called RDDs [5]. Spark is much faster than its predecessor Hadoop(cite) because it can pack multiple operations into a single task, and uses the main memory more efficiently insead of reading and writting the intermediate data to the disc.

Contributions

- We provide a new RDD extension, that helps the user express hierarchical structures.
- We evalutate that RDD extension using the cluster with 5 nodes provided by WIND.

We run experiments with synthetic datasets consisting of millions of points, and found that using the hierarchical RDD with Scala parallel collections, total processing time can be reduced by up to 40%.

3.2 Design

The main data abstraction in Spark is Resilient Distributed Datasets (RDDs). RDDs represent an immutable collection of data, partitioned and distributed across the cluster. That data can be processed in parallel according to the number of the dataset partitions. The RDD API gives a collection of operators on the dataset (including map, filter, groupByKey, cache, persist), enough to express a wide variety of applications. Note that the RDD operators are lazy, meaning that they do not compute the results right away. Instead, Spark creates a pipeline of transformations and evaluates it explicitly when the collect operator is invoked.

The existing RDD representation and operators work well with a wide set of problems and algorithms, operating on "flat" data collections, such as map-reduce programs. There are, however, algorithms and computations that require structuring the data set in different ways. For example, the data decomposition in a divide-and-conquer algorithm or the hierarchical back-tracking of a dynamic programming algorithm require the programmer to create complex structures of RDDs that capture the "non-flat" structure of the data.

For example, consider the divide-and-conquer algorithm for computing hierarchical K-Means clustering presented above. Note that the data is initially "flat", but the algorithm discovers and maintains structure during the computation.

Expressing such a hierarchical algorithm with the existing RDD operators can be quite challenging for the users. The splitting of the data in many levels results in a tree of RDDs, that are quite difficult to handle and maintain. Also nodes in the same level of the tree represent disjoint tasks, that can be issued in parallel.

3.3 Hierarchical RDDs

We present a new RDD abstraction that helps the programmer create a tree collection of RDDs and issue independent jobs in parallel. To make the Spark RDDs more expressive for hierarchical structures, we created an RDD extension, called hierRDD, and use hierRDD to encode bisecting k-means in a much more forward and intuitive way, while also improving execution performance.

In order to create hierarchical RDDs the user should first provide an object that implements the Splittable interface described in Figure 4. The Splittable object repserents a hierarchical structure that can be splitted into smaller sub regions. Thus the user should implement the function *contains* that specifies whether an element is contained into the Splittable object, and the *Split* function that returns an array of the subregions the object is splitted. The SplitPar method is identical to the Split method, except that it returns a Parrallel Array, so that the Spark driver can issue the jobs concurrently.

An example of implementing the Splittable trait is the Cluster class shown in figure 5, that is later used to code the bisecting K-means algorithm. The constructor takes as arguments the identity of the cluster, and the number of iterations (k-means specific). To split the initial data we use the KMeansModel from the Spark MLlib library. The m variable represents the clustering

```
1 def hierarchical[A:ClassTag](s:Splittable[T]) = {
2     new HierRDD(this,s)
3 }
4 trait Splittable[A] {
5   def id : Int
6   def contains(a:A) : Boolean
7   def splitPar(level:Int) : ParArray[_ <: Splittable[A]]
8   def split(level:Int) : Array[_ <: Splittable[A]]
9 }
```

Figure 4: API for creating hierarchical RDDs

model, used to define to which subcluster each point belongs. The split method iterates through the cluster center and for each one it creates a new Cluster instance with the id of the cluster.

Figure 6 describes the main loop in the bisecting K-means application. Line 1 creates the initial cluster that contains all the data elements(that implements the Splittable interface), and then in line 2 we create a hierarchical RDD from the data. The while loop in lines 5–8 continiously splits the cluster into smaller subclusters until we reach the desired number. The splitPar operator returns a ParArray(scala.collection) so the splitting is issued in parallel, resulting in reduced total time compared to the sequential one.

3.4 Evaluation

To evaluate the performance of hierRDD, we use the cluster with 5 machines provided for ASAP in the WIND data center. Each machine has 4 i5-3470S cores, 16G RAM memory, and 2.4TB SSD, running Debian linux. The Spark architecture includes one master and 4 slaves. Each slave creates 2 worker instances, and assigns 2 cores to each one. Totally the workers are 8. The dataset is some randomly generated points of 20 dimensions, that are stored in a HDFS file system. After we call the textFile method, we cache the points RDD in order to achieve better locality. We evaluated our design comparing the hierRDD, hierRDD with parallel splitting, and the default implementation using simple RDDs and MLlib. We run the experiments with 2, 4, 6, and 8 slaves to measure scalability.

The fitst data-set contains 1 million points, of 20 dimensions each. Table 3 shows the time in seconds for each K-means variation, for different number of slaves. The last column measures the speed up gained from hierRDDpar compared to hierRDD. For the maximum number of workers the speedup is 40%. The second data-set has 2 million data points. Table 2 shows the time scale and the speed up. For 2 workers hierRDDpar gains speedup 10% compared to hierRDD and 37% for 8 workers.

Parallel hierRDD gains speedup over sequential hierRDD because the cluster utilization is higher and the load imblance between different tasks is mitigated.

```
1  class Cluster(id: Int, iter:Int)
2    extends Splittable[Vector] with Serializable{
3
4    def id() = id
5    var data:RDD[Vector]
6
7    var m : KMeansModel = KMeans.train(data, k=2, iter)
8
9    def contains(point: Vector) = (m.predict(point) == id)
10
11   def split(level:Int) : Array[Cluster] = {
12     m.clusterCenters.zipWithIndex.map{ case (c, idx) =>
13         new Cluster(idx, iter)
14     }.toArray
15   }
16
17   def splitPar(level:Int) : ParArray[Cluster] = {
18     split(level).par
19   }
20 }
```

Figure 5: Splittable subclass implemented for the Bisecting K-means benchmark

```
1 val initcluster = new Cluster(id=0, data, subIterations=3)
2 val hierrdd = data.hierarchical(initcluster)
3
4 var split = hierrdd
5 for(i <- 1 until maxdepth){
6   split = split.flatMap( subrdd => subrdd.splitPar())
7 }
```

Figure 6: Bisecting K-means implementation with hierarchical RDDs

workers	hierRDD	hierRDDpar	mllib	speedup
2	2475	2248	2542	1.10
4	1418	1156	1394	1.22
6	1124	877	1101	1.28
8	872	636	858	1.37

Table 2: Time table(seconds) for 2 million data points

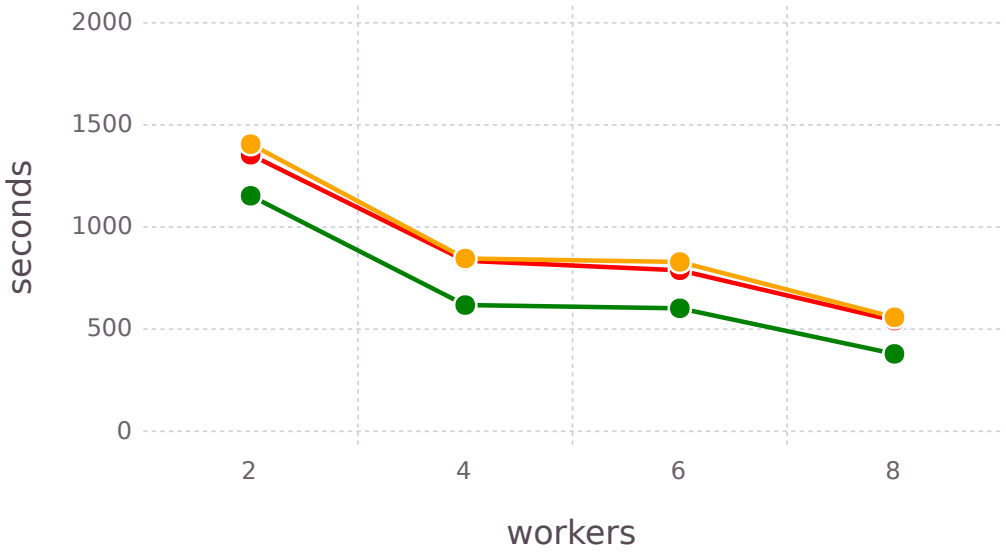

Figure 7: Strong scaling graph for 1 million data points

Figure 8: Strong scaling graph for 2 million data points

Figure 9: Weak scaling graph

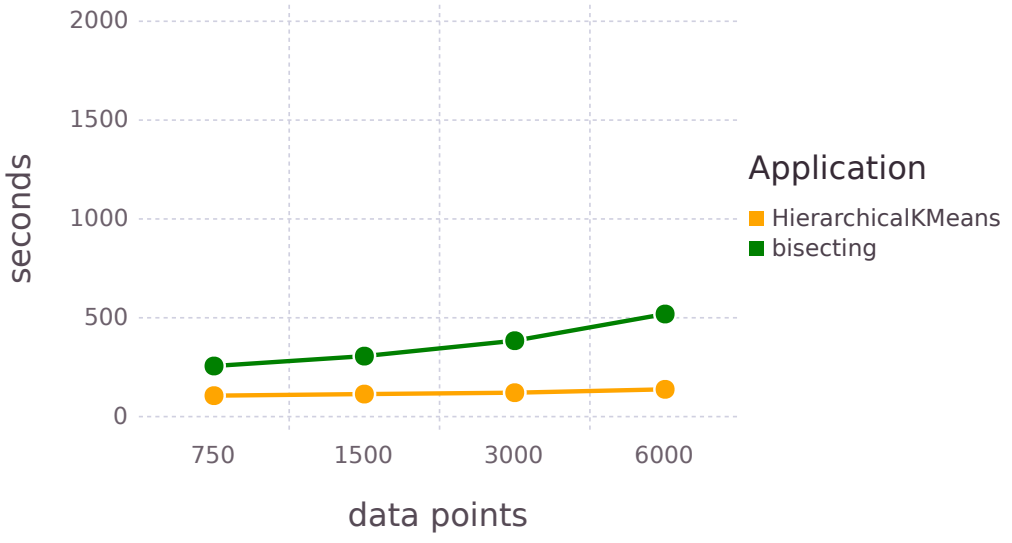

Figure 10: Naïve version versus hierarchical RDDs

workers	hierRDD	hierRDDpar	mllib	speedup
2	1406	1153	1354	1.21
4	846	618	835	1.36
6	829	602	788	1.37
8	558	379	542	1.42

Table 3: Time table(seconds) for 2 million data points

npoints	hierrdd	bisecting	speedup
750	106	256	2.42
1500	114	306	2.68
3000	121	384	3.17
6000	138	519	3.76

Table 4: Time table(seconds) comparing hierRDD with naive solution

To expose the expressiveness of our hierRDD implementation we compared our hierarchical K-means variation with one using the default RDDs found in `https://gist.github.com/freeman-lab/5947e7c53b368fe90371`. Figure 10 shows the time elapsed for both applications for various data points. The figure shows that for 750 points our implementation is 2.4× faster, and that increases to 3.7× for 6000 points.

Figure 9 is a time plot for various data points, from 3000 to 96000, which means the lower value the better, for depth 8 (the leaves have 256 nodes), utilizing 5 slaves (10 executors). Three versions of the bisecting k-means are compared, the default algorithm without the hierarchical RDDs, while the other two use the hierarchical RDD abstraction, one with sequential job issue and one with parallel job issue. The results show that the hierarchical RDDs implementation incurs zero overhead when done sequentially.

4 Scalable distributed scheduling

In cluster mode, Apache Spark is deployed as a master-slave architecture. The main software components of the Spark scheduler are SparkContext, DAGScheduler, TaskSchedulerImpl, and CoarseGrainedSchedulerBackend. SparkContext is created by the user application, and is responsible for creating RDDs from files or local structures, and submitting the jobs to master. Figure 12 shows a general map of the scheduling path.

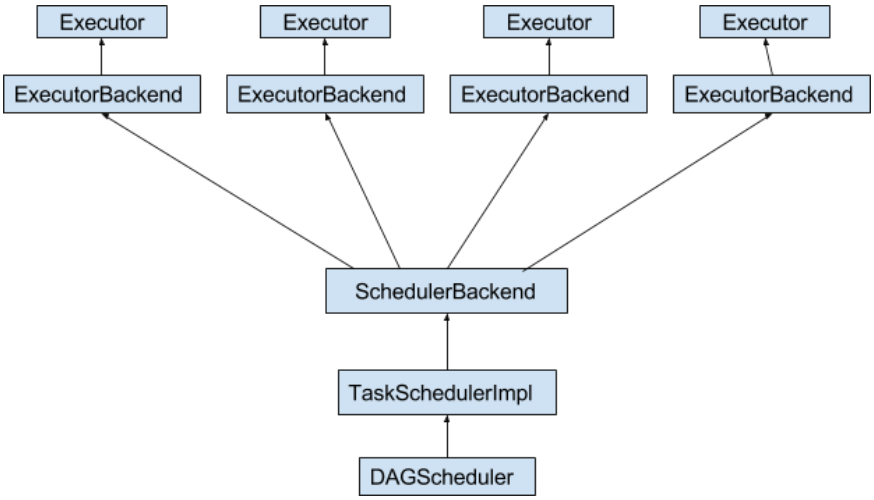

Figure 11: Spark Cluster Mode Architecture

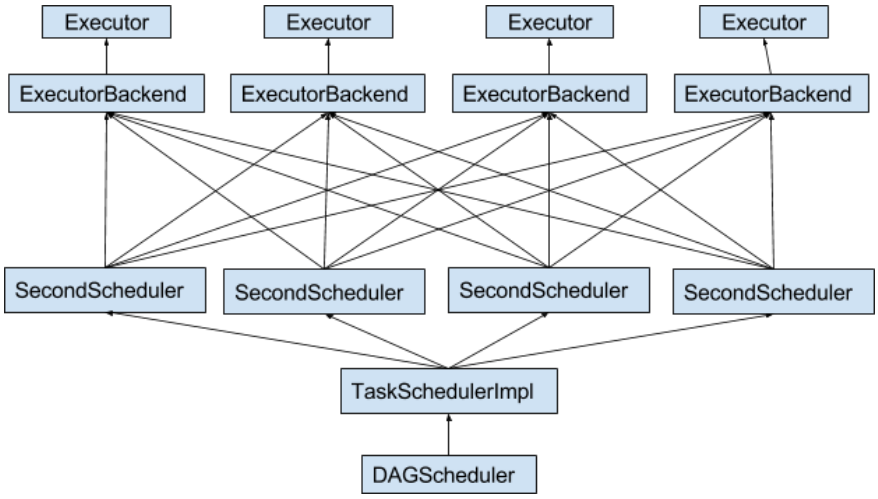

Figure 12: Parallel Scheduling Architecture

```
1 val array = Array.tabulate(100000)(i=>i)
2 val rdd = sc.parallelize(array).repartition(10000)
3 val sum = rdd.map(rdd => rdd.reduce(_+_))
```

Figure 13: Many tiny tasks micro benchmark

4.1 DAGScheduler

Jobs are submitted to the DAGScheduler actor. Those jobs are broken into stages, organized as a graph, that are submitted as TaskSets for execution. Each job creates as many tasks as the number of partitions of the dataset it operates on. DAGScheduler serializes and broadcasts the job to every worker. Additionaly, it holds the task metadata, regarding the number of attempts, preferred locations, and stage dependencies.

4.2 TaskSchedulerImpl

TaskSchedulerImpl is responsible for monitoring the workers. It receives the heartbeats from each worker, and handles the StatusUpdates messages, that refer to either the success or the failure of a task. During the submission of a TaskSet, TaskSchedulerImpl maps the tasks into the resources, according to the task localith preferences.

4.3 SchedulerBackend

SchedulerBackend maintains the necessary information for the workers in a data structure, that contains the address, the number of cpus, and the current load of the worker. The main job of the SchedulerBackend is the launchTasks function, that sends the task to the locacion specified.

4.4 Motivation example

Figure 13 shows a case where spark's default scheduling mechanism lacks effiency, because

1. The scheduling path is sequential, which means that if the job consists of many tiny tasks the scheduling process will take a lot of time while the processing time will be negligible.

2. After the worker has finished a task, it has to send a StatusUpdate message to the scheduler, so that the worker invokes again a makeOffers call to send a new task to the worker. That increases the total time by at least one RTT.

To solve such cases, as well as cases where a large number of executors cannot be properly managed by a single scheduler, we design and implement a distributed version of the Spark scheduler.

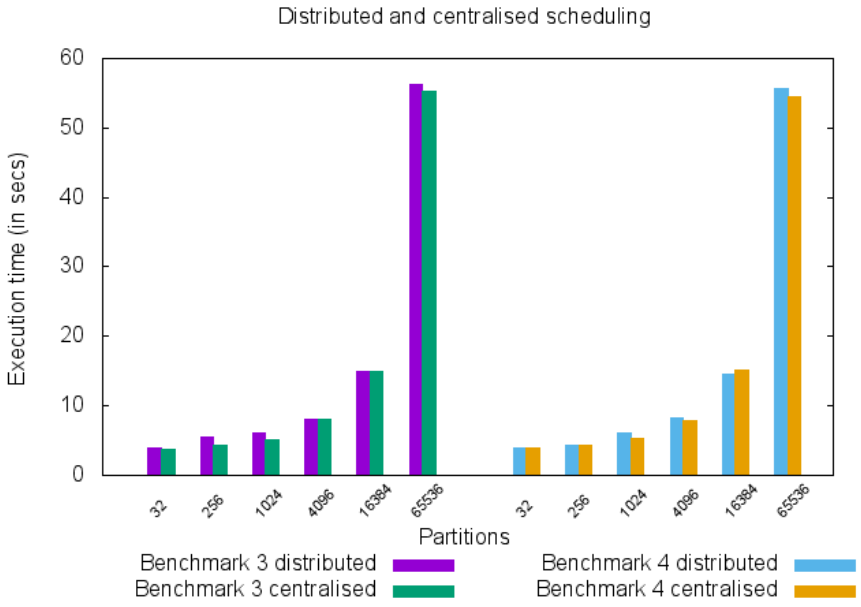

Figure 14: Preliminary results of the distributed scheduler performance

4.5 Distributed Scheduling

Our first goal is to lower the gap that is created by the time the worker finishes the task the the time he receives the next one to run. To accomplish this, we inserted a local task queue per worker. Every time a worker finishes a task, he looks up the task queue if there are more tasks. This look up is much cheaper than getting a task from the driver, because no network traffic is generated.

Second, we parallelized the sending of the tasks to the workers within a single TaskSet. A TaskSet contains a sequence of tasks, all referring to the same job. Instead of using a single SchedulerBackend, we have a set of schedulers, namely SecondLevelSceduler actors. The TaskSet is partitioned into smaller sets, each one sent to a scheduler that is then responsible for sending the individual tasks to the workers. The scheduling of a single task by each scheduler is simple; the scheduler picks a random worker and sends the task, no locality considered. Also, multiple tasks can be sent at once under load, so that the worker queues are full.

4.6 Benchmarks

We are currently implementing the above design for distributed scalable scheduling in Spark. We have evaluated the current early implementation using a set of microbenchmarks with very small tasks designed to stress the Spark scheduler. Figure 14 presents a comparison on microbenchmarks

between the existing Spark scheduler and the distributed scheduler described above. Although preliminary results do not show a measurable difference, note that this experiment was run on a 5-node cluster on which one scheduler node suffices almost always to keep the remaining 4 worker nodes busy. We present the preliminary results for completeness here and expect more meaningful results on larger clusters in the future.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

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

[3] Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In *In KDD Workshop on Text Mining*, 2000.

[4] Wikipedia. Latex — wikipedia, the free encyclopedia, 2015. [Online; accessed 23-October-2015].

[5] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D4.2
Execution Engine v.1

WP 4 – Dependency-aware query execution engine

Nature: Report

Dissemination: Public