Pipelined execution of stages in Apache Spark

(extended abstract of the MSc dissertation)

Truong Duc Kien
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Paolo Romano

***Abstract*—This dissertation investigates the efficiency of a fundamental, low-level building block of modern big-data processing platforms, like Apache Spark. This type of platforms supports complex data analysis tasks by allowing data scientists to express arbitrary data manipulations via a direct graph of, so called, transformation operators that executed in a distributed fashion across a cluster of machines. In many realistic use cases, it is necessary to recur to operators, like Reduce, which have to fed with the results produced by operators executing on different machines. Some state-of-the-art solutions, like the aforementioned Spark platform, take what can be defined a batch strategy, which operates as follows. Operators that induce exchange of data among machines are used to demarcate the beginning of a new *stage*, which serves to logically group a set of operators that can execute without requiring any input from operators running on different machines. A Spark application is executed in batch mode, in the sense that if the stage $i + 1$ requires the result of stage i, then stage i needs to finish its computation before stage $i + 1$ can begin and load the data from stage i. This choice allows for a set of benefits, including higher network usage efficiency, simpler scheduling over operators and fault-tolerance mechanism. However, it can also lead to sub-optimal utilization of cluster's resources. Another alternative approach consists of pipelining data transmission between stages, so that stage $i+1$ can start processing the data as it is generated by stage i. With this strategy, the execution of the two stages are overlapped, allowing higher utilization of cluster's resources. In this thesis, Apache Spark is extended to support the pipeline execution of stages in order to compare the two strategies. An extensive experimental evaluation was conducted using both synthetic and realistic workloads to evaluate the performance difference when switching from batch to pipeline execution.**
Keywords: data-analytics, Spark, big-data, batch, pipeline

I. Introduction

A cluster computing framework composes of two layers: the application programming interface (API) and the underlying execution engine - also called the runtime. In order to use the framework, a developer has to write applications that contain customized logic to manipulate data using the provided API. Some frameworks also provide predefined functions for common tasks such as sorting, grouping, or filtering data to ease development effort. Both the user-defined functions and the provided utilities will be referred as **operators** in this document. A data analytic job is implemented by combining these operators in a coherent order. This order specifies the dependencies between operators: an operator can consume the output of different operators, and in turn generates input for other dependent operators.

Once a program is created using the data analytic framework's API, it is executed on the cluster by the framework's execution engine. As the data is processed, it needs to be transferred from an operator to another. Apart from a few embarrassingly parallel workloads, most programs require data to be exchanged not only between operators in the same machine, but also between operators running on different computers in the cluster.

In many state-of-the-art platforms, such as Map/Reduce [1][2] and Spark [3], when an inter-computer data transfer is required, it is performed by a two-stage batch shuffle operation, where the output of one operator are partitioned and sent to other computers in the cluster based on the key of each element in the output. As shown in figure 1, during the first stage of a two-stage batch shuffle, all sending operators are required to store shuffle data to a buffer, and a synchronization barrier is used here to ensure the completion of all senders. In the second stage, the data is fetched and processed by the receiving operators. Due the existence of the barrier between stages, the faster machines in the cluster have to wait idly for the slower machines to catch up.

An alternative to the batch shuffle, shown in figure 2, is the pipeline shuffle, where the records are transferred from the sender operator to the receiver operator as they are processed, without having to wait for the sender operator to finish. This design does not require the use of a stage barrier, and are often used in real-time streaming systems like S4 [4], Storm [5], and Flink [6], where the processing time of each record have to meet a strict deadline. Still, in other use-cases, the batch shuffle used in Map/Reduce and Spark offers many advantages over a pipeline system.

In the first place, because the output of a stage is grouped into large chunks before transmitting to other computers in the cluster, the number of network connections and their overhead are minimized. Furthermore, the transferred data can also be compressed with a higher ratio than in a pipeline system, reducing the amount of data needed to send over the wire.

Moreover, being able to capture a snapshot of the system makes implementing fault-tolerance much easier. In a batch shuffle, a checkpoint can be performed by persisting the

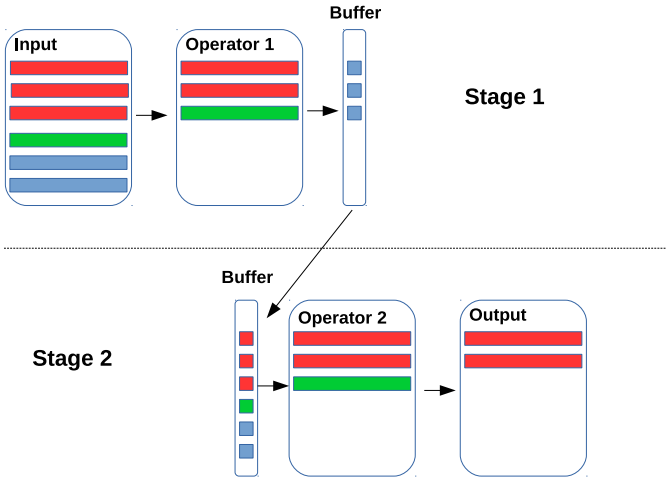

Figure 1. Two-Stage Batch Data Transfer. The red rectangles represent already processed records, while the currently-processed records are colored green. The blue rectangles indicate unprocessed records.

output buffer to disk, as done in Spark. In a pipeline shuffle, it's much more difficult to do a snapshot, since the state of the network channel between computers also needs capturing.

Lastly, a batch shuffle system completely decouples the sending and receiving of data, thus does not require the producer and the consumer stages to be run at the same time. This simplifies the scheduler, because only the operators in the current stages have to be factored in the calculation. In a pipeline shuffle, the scheduler have to make sure that the operators in both the producer and the consumer stages are active at the correct order to avoid deadlocks. In applications that have many shuffles, where a stage can be both a producer and a consumer of shuffle of data, a pipeline system will require the scheduler to create a scheduling plan for all operators in the application at once.

Despite all the advantages of a batch shuffle system, the presence of a stage barrier wastes CPU times in faster computers and reduces system's efficiency. Due to internal and external factors, even identical machines can perform differently in a cluster, and faster computers will have to wait at the barrier until the slower ones finish. This is particularly cumbersome in cloud computing environment where a computer cluster are often made up of virtual machines (VMs), which naturally receive a higher degree of interference due to the co-location of multiple VMs on the same physical host. As a Spark program might contain multiple synchronization barrier, it can waste a lot of CPU times in such environment. The goal of this thesis is to investigate the impact of using a pipeline execution strategy in Apache Spark by removing the stage barrier, both in terms of implications on correctness of the computation and on fault-tolerance. Furthermore, via an extensive experimental evaluation using both synthetic and realistic workloads, we measured the performance difference of Apache Spark when switching from batch to pipeline execution.

II. Background

A. Brief history of big-data analytic platforms

Cluster computing is not a recent invention: initial efforts to use multiple computers working in coordination to solve problems, which cannot be solved by one computer, appeared in the 1960s [7]. However, it was not until the 1980s that a general purpose computer clusters become commercially successful with the appearance of DEC's VAXcluster system [8]. Further development to standardize and simplify the usage of large scale, and potentially highly heterogeneous clusters leads to the advent of MPI - Message Passing Interface [9] and PVM - Parallel Virtual Machine [10] in the early 1990s. These two solutions still remain widely used today in high performance computing.

Despite being the de-facto standard in cluster computing for a long time, the MPI standard was quite low-level, difficult to use and has no mechanism to provide fault-tolerance. This makes MPI only suitable in controlled environments where computers and network are extremely reliable, which are expensive and difficult to build and maintain. Over the last decade, we have seen a proliferation of cluster computing platforms, like Map/Reduce [1] and Dryad [11], aimed at simplifying the development of large-scale data analytic applications to run on cheap, unreliable, commodity hardware. The rise of cloud computing, an even more unreliable environment, further increases the necessity of these new data processing frameworks.

B. Modern big-data analytic platforms

Map/Reduce [2] [1] might be the most well-known modern data processing framework. The execution of a Map/Reduce program is divided into two distinct stages. During the first stage, the data is distributed to all machine in the cluster to process and generate results in the form of key-value pairs. In the second stage, results are grouped by key and all pairs with the same key are collected and analyzed at the same computer. The users only have to supply one function to run in each phase to work on the input, while everything else is handled by the framework. This is much simple than writing a similar program using MPI or other low-level tools, although at the cost of having to adopt a less expressive programming paradigm.

Dryad [11] is another cluster computing framework developed by Microsoft. A Dryad application is modeled as a directed acyclic graph (DAG) in which the computational tasks are performed in the vertices and the communication channels are represented as edges. Compared to Map/Reduce, Dryad is highly flexible. Firstly, a Dryad application can specify a variety of communication patterns, while Map/Reduce is restricted to the scatter - gather topology. Secondly, a Dryad application can have many operators, each can have multiple input and output channels. On the other hand, a Map/Reduce only have two operators with strict input and output requirements. This flexibility mean many algorithms are easier to implement in Dryad than in Map/Reduce.

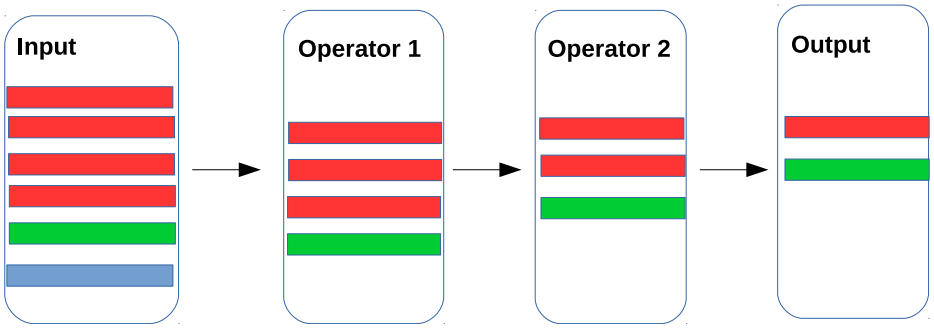

Figure 2. Fine-grained Pipeline Data Transfer. The red rectangles represent already processed records, while the currently-processed records are colored green. The blue rectangles indicate unprocessed records.

S4 [4] is a general-purpose, distributed scalable, and fault-tolerant stream processing platform developed by Yahoo!. Its main objective is to provide an easy way to develop application that can process continuous, unbounded stream of data. The data in S4 system is represented using events, in the form of key-value pairs - similar to Map/Reduce. The operators in a S4 application are called the *Processing Elements* (PE), which are implemented by the programmer. The system will automatically initialize one processing element for each key to process all events with that key. On receiving and processing an event, the PE can either emit more events for further processing by other PEs, or publish the result to external systems. The main advantages of S4 over Map/Reduce are lower latency and the ability to handle unbounded data, thanks to its pipeline design.

Storm [5] is also a distributed stream processing engine, similar to S4, but with some major improvements. Firstly, while a S4 data stream consists of key-value pairs, a Storm data stream is made of tuples, which offers more flexibility. Secondly, the operators in Storm, called bolts, have to handle unrelated tuples that exist in a stream of events: there are no group-by-key performed by the framework like in S4. In addition, the number of operators have to be configured by the users and not automatically determined by the framework. Thirdly, in Storm, an operator pulls the event from the source, while S4's operators push the events to the receiver. This difference means a Storm application is less vulnerable to buffer-overflow errors like in S4. Finally, Storm provides guarantee about events processing: either "at-least-once" or "at-most-once" depending on the user's requirement, while there is no such guarantee in the case of S4.

Spark [3] started out as a research project in UC Berkeley but has grown to become one of the most popular cluster computing framework in recent years. This is thanks to its high performance and user-friendliness. Spark has one of the easiest to use API, where the developers can use customized functions to manipulate data and elegantly chain the output of these functions to specify data dependencies. This simplicity speeds up Spark adoption significantly. In term of performance, Spark is a major improvement compared to Map/Reduce. Instead of chaining tens of Map/Reduce programs to implement an algorithm, and paying the expensive price of disk I/O, the developer can implement the entire logic of the algorithm in a single Spark program. In addition, Spark runtime tries to keep the data in resident memory for most transformations, thus significantly reducing the amount of disk accesses. Consequently, algorithms implemented in Spark tend to be much faster than their Map-Reduce counterparts.

Originally designed as a batch processing engine to replace Map/Reduce, Spark has expanded to include stream processing capability. Because the runtime of Spark uses a batch design for shuffling data, an application is divided into multiple stages. These stages are arranged in to a directed acyclic graph and executed topologically. This separation of stages means that the program will never progress pass the first stage without a limit on the input data, making Spark unsuitable for streaming workload. In order to solve this problem, Zaharia et al. proposed a system called **Discretized Stream (DStream)** [12], in which the unbounded input stream is split into smaller batch that can be processed sequentially using the original Spark runtime. This approach is also called micro-batching, because the batches have to be small enough so that the system can run them in seconds to satisfy the stringent latency requirement of stream processing. Once the result of each micro-batch is calculated, the programmer can easily merge them together to calculate the final result if necessary. Nonetheless, DStream still used batch execution, so the cluster's resource is still wasted at the synchronization barrier between stages.

C. Apache Spark Fundamentals

As already mentioned, the target platform used in this dissertation is Apache Spark. In the following I provide an overview about the high-level design of Spark, and its basic programming interface.

The core of Spark is based on the concept of *Resilient Distributed Datasets* (RDDs), a distributed memory abstraction that allow programmers to perform fault-tolerant data processing across an entire cluster of computers. An RDD is an immutable set of records and can only be created through deterministic *transformations* from either the input data or other RDDs. As a result, as long as the input data

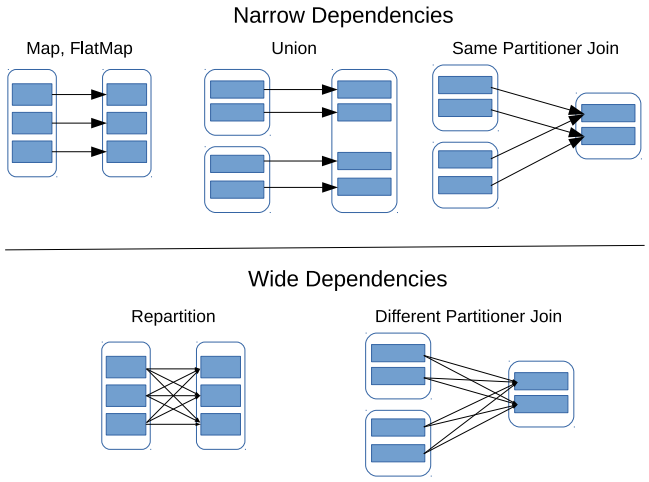

Figure 3. Spark's dependency types: blue rectangles represent the partitions of a RDD, while the arrows indicate dependencies.

is stable, the content of a RDD is specified only by these transformations. Thanks to this design, Spark is able to provide fault-tolerance purely by remembering the series of transformation that is used to generate each RDD. This series is also called the *lineage* of an RDD, and is used to re-construct the RDD in case of failure.

Apart from *transformations*, the other type of data operator in Apache Spark is called *actions*, which trigger processing of the RDD to return results to the driver program or making the RDD content available to external systems. A Spark program is always lazily-evaluated: each RDD is never materialized immediately where the transformation is specified in the driver code. Instead, the occurrence of a transformation causes the lineage of the RDD to be updated with the new transformation. Only when an action is called on an RDD, will Spark trace that RDD's lineage and perform all transformations needed to compute it.

As mentioned, Spark's transformations are used to create new RDDs from old RDDs, which introduces dependencies between the input and output RDDs. In Spark, these dependencies are divided into two groups: **narrow dependencies** and **wide dependencies**, which are illustrated in figure 3

Narrow dependencies happens when each partition of the parent RDD is only used to compute at most one partition of the child RDD. This type of dependencies are very efficient because Spark will co-locate the parent and child partition in the same computer and use shared memory to pipeline the data directly without going through the shuffle.

Contrary to narrow dependencies, wide dependencies means each partition of the parents RDDs is needed by more than one partitions of the child RDD. For example, when you do a *reduceByKey* transformation, as the key can exist in any partitions of the parent RDD, all of these partitions are needed to generate a partition of the child RDD. Similarly, when joining two RDDs which were partitioned using different partitioners, they have to be repartitioned with the same partitioner so that the items with the same key in both RDD are gathered in the same machine. Consequently, a wide dependency always requires data shuffling and is also called a shuffle dependency.

III. Implementation

A. The architecture of Spark

Spark uses a master/workers architecture: each Spark cluster has one master node and multiple worker nodes. The master node is connected to all worker nodes to manage the cluster's resource and provide the single gateway into the clusters, while the worker nodes perform the necessary computations of the data analytic job.

Each Spark's application has a driver program, which can be run in a machine outside the cluster or in a worker node. This driver application is written by the programmer to implement the processing logic. Once started, this driver connects to the master, and get assigned resources of the worker nodes in the form of executors processes.

The executor process is the muscle of a Spark application: it receives tasks from driver, runs those tasks and reports results and other statistics back to the driver. Executors are spawned in worker nodes on request from the master and generally live for the entire application. During its lifetime, the executor will generally run multiple tasks as assigned by the driver.

B. The components of Spark

1) DAGScheduler: From the driver program written by the programmer, Spark creates an execution plan consists of multiple jobs, one job for every *action* performed. A job is created by grouping all *transformations* performed from the beginning of the program up to that *action*. Once the job is generated, it is converted into a directed-acyclic-graph (DAG) and executed by the *DAGScheduler* components of the driver.

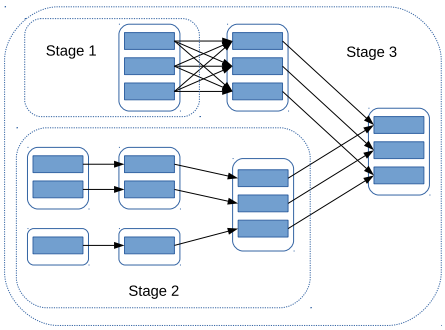

Figure 4. Spark's DAG construction. A blue rectangle stands for a partition of a RDD, which is represented by the surrounding rectangle. The dash-lines are used to separate the stages.

Once the DAG is created, Spark will start executing the stage in topological order. For example, with the program in figure 4, stage 1 and stage 2 are always executed before stage 3. However, stage 1 and stage 2 can be executed parallelly or serially, because there is no dependency between them.

The final job of the *DAGScheduler* is to handle failures due to the lost of map output files, in which case the old stages need to be run again to regenerate those files.

2) TaskScheduler: For each about-to-be-run stage, Spark generates a set of tasks and pass it to the *TaskScheduler* components, which will allocate them to the executors.

A task is the smallest execution unit in Spark. Each task is fully independent, self-contained, and identified by the stage id, the stage attempt id and the partition id. In other words, a task represents the computation of a single data partition in an attempt to execute a stage in a Spark job.

Inside the task scheduler, the set of tasks for each stages is managed independently using a separate *TaskSetManager*. A *TaskSetManager* keeps track of each task in the set and matches the tasks with the preferred executors using an algorithm called delay scheduling [13]. This technique allows a small delay before a task is matched with an executor in order to improve the matching efficiency, i.e the task is best run on an executor that has already contained a large amount of its input data.

In addition to pairing tasks and executors, a *TaskSetManager* also watches for tasks' failures and retry them up to 3 times if possible.

3) CoarseGrainedSchedulerBackend: A Spark's scheduler backend acts as the executors' repository for a Spark application. It tracks executors' registration, de-registration, and maintains a record of free executors and CPUs across the cluster. In this thesis, we focus on the *CoarseGrainedSchedulerBackend*, which is used by Spark's standalone cluster and is the most popular one. This scheduler backend also reuses the same executors for the entire duration of a Spark application, instead of launching a new executor for each task.

4) MapOutputTracker: The *MapOutputTracker* is the component that tracks the location of shuffle output produced by a stage. For each stage, the map output is divided into buckets, each bucket is generated by a task and corresponds to a partition in the input RDD. There are two versions of *MapOutputTracker*: one run on the driver call *MapOutputTrackerMaster* and one run on the executor call *MapOutputTrackerWorker*, the later acts as a local cache for the map output's location stored in the former.

Whenever a task finish processing, it stores the output in the local block manager and contacts the driver, which will register the location of that task's map output in the *MapOutputTracker*. A map output location is defined by the shuffle's id, the partition's id and the block manager's id. Afterward, when the tasks of a child stage are launched, they will contact the *MapOutputTracker* to get the location of the parent stages' map output so that they can fetch and process this data.

C. Extending Spark for coarse-grained pipelined stage execution

While changing Spark's runtime to use a fine-grained pipeline execution strategy, like in Storm or Flink, could give us a more contrasted comparison to the original batch execution strategy, it would be a very challenging tasks.

In the first place, it is impossible to fetch a task's output before that task finished entirely, because Spark's shuffle process is based on sorting. The output of a task need to be sorted and indexed before persisted to disk, so that the consumer tasks can fetch an entire region of shuffle file without having to do a full scan. In order to do a fine-grained pipeline shuffle, the shuffle module will need changing to a hash-base approach, so that each record can be forwarded directly to the correct consumer task based on the hash of its key.

In addition, in a fine-grained pipeline system, all consumer tasks have to be schedule to run concurrently with all producer tasks, otherwise the generated shuffle data cannot be pull by consumer from producer. In a Spark program, all stages are either producers or consumers, so all stages need to be scheduled to run concurrently. This will require a very large change to the scheduler.

Furthermore, one of the largest advantages of pipelining is the ability to reduce disk I/O by writing little to no data to disk. A pipeline shuffle, which write all data to disk, is self-defeating. However, Spark uses the on-disk shuffle files as part of its fault-tolerant mechanism. As a result of this conflict, a switch to a fine-grain pipeline shuffle also necessitates a new mechanism to recover from failure in computation.

For these reasons, a coarse-grained pipeline approach, inspired by [14], was chosen to evaluate in this thesis. The shuffle component remains unchanged: the output of each task of the producer stage is still sorted and written to disk, and the tasks of the consumer stage still fetch data in large chunks instead of doing it record-by-record. However, Spark is modified so that the cluster does not wait for the producer stage to end before launching the consumer stage. On the contrary, some tasks of the consumer stage are started early to process the shuffle data even when they are incomplete. Furthermore, the consumer stage's tasks are modified to wait for the currently running producer tasks when they have exhausted all available shuffle data.

This design allows the two stages to overlap, and takes advantage of the resource that would be wasted if the cluster waits for producer stage to end. Furthermore, because the shuffle data is still persisted to disk, the fault-tolerant mechanism of Spark is not compromised.

1) Early scheduling of the consumer stage: Since the consumer stage is dependent on the shuffle output of the producer stage, the tasks in the former will never finish until the later is done. Thus, we need to make sure that all the tasks in the parent stage are either already finished or currently running before we attempt to start the child stage. This requirement will guarantee that the child stage's tasks will not occupy up all the CPU slots, and thus starve the parents stage, causing deadlock.

In order to implement this, we add a method into the *CoarseGrainedSchedulerBackend* to check if there are enough free CPU cores to execute all pending tasks. When-

ever a task finishes, the *DAGScheduler* will use this method to check if it is safe to start the children stages, which is when the number of all pending tasks are less than the number of free cores.

2) Processing incomplete shuffle data: In an unmodified version of Spark, when a task in the consumer stage starts executing, it contacts the *MapOutputTracker* to get the list of shuffle data's locations that was produced by the producer stages. As all parents of this consumer stage have completed, the shuffle data have already been stored in executors' block manager. Thus, the task can safely contact the block managers which house the shuffle data and retrieve it for processing.

However, this assumption no longer hold in our modified version of Spark, because the child stage can be started before the parents stages have finished. Consequently, the consumer's tasks have to deal with incomplete shuffle data. In order to fulfill this requirement, we have to modify how the shuffle data location is saved and how it is read.

Firstly, the *DAGScheduler* is modified to update the *MapOutputTrackerMaster* with new shuffle location every time a task finished, as opposed to update only when the entire stage is completed in the original code. This change allows a consumer stage to always have some data to process when it is started, even if the producer stages have not finished.

Secondly, the *MapOutputTrackerWorker* in the executors is set to periodically update itself with new shuffle data locations from the driver, as long as the cached data contains empty locations. This tasked is performed by a dedicated *MapOutputUpdater* in a background thread. This updater is started when the first task in the consumer stage contacts the executor's *MapOutputTrackerWorker* for shuffle data, and continues running until the location information of the shuffle is complete.

Thirdly, changes has to be made to how the shuffle data is processed. Traditionally, Spark implements shuffle computation using the volcano iterator model [15] as shown in figure 5. The bottom layer, *ShuffleBlockFetcherIterator*, retrieves shuffle blocks from block manager locally or remotely, decodes, and exposes the data as an iterator of records. Each consumer's task has its own iterator, and uses it to get records for processing.

We modify this design and add another layer, *PartialShuffleBlockFetcherIterator*, to handle missing data blocks. The *PartialShuffleBlockFetcherIterator* is started with a *ShuffleBlockFetcherIterator* seeded with all the available shuffle data at that moment. As soon as this initial iterator runs out, the *PartialShuffleBlockFetcherIterator* will contact the local *MapOutputTrackerWorker* to get the list of new shuffle blocks that have become available while the initial data was processed. Afterwards, a new *ShuffleBlockFetcherIterator* is created by the *PartialShuffleBlockFetcherIterator* to handle this new data. This process repeats until the producer stage completes and all shuffle data becomes available. The modified iterator model is presented in figure 6

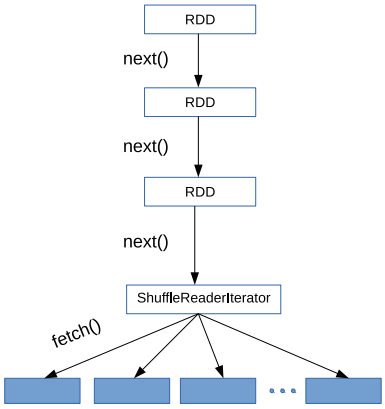

Figure 5. Shuffle fetcher iterator: each blue rectangle represents a partition of shuffle data

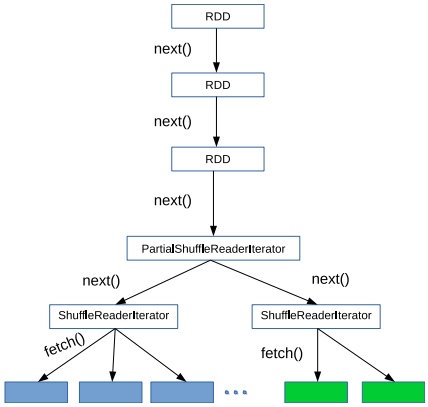

Figure 6. Partial shuffle fetcher iterator: the blue rectangles are the partitions of shuffle data which are ready when the stage is started, while the green rectangles indicates the partitions which become available after that.

3) Over-scheduling executors: While doing our experiments, we noticed that during shuffle, the map output can be consumed a lot faster than it is produced. As a result, even when the consumer's tasks are launched early, they spent a lot of time waiting for the shuffle data to be generated. Instead of letting the CPUs go to waste while the tasks are idle, we decided to launch additional tasks while the first few ones are waiting for more shuffle data. This is implemented by adding a hook inside the *PartialShuffleBlockFetcherIterator* to notify the executor when the initial shuffle data is completely processed and the task needs to block and wait for new data. On receiving this signal, the executor will attempt to launch an additional task, while the first task is waiting for more data. However, because memory requirement grows linearly with the number of tasks, launching too many tasks will cause the executor to quickly run out of memory. In order to avoid that situation, by defaults we do not launch more than one additional task for every early-start task running on an executor.

D. Fault-tolerance implications

As presented, Spark provides fault-tolerant computation by recalculating the RDDs based on their lineage. However, tracing the entire lineage from the first transformation can potentially take a lot of time, especially when data-shuffling is involved. As a result, Spark takes advantage of the on-disk shuffle files as a check-point mechanism. Because these files are never deleted until the program finished, whenever a task in a stage need to be restarted due to some execution errors, they can re-read the input shuffle files to recompute the corresponding partition. This retry is performed transparently up to three times by the TaskScheduler, and we don't have to make any modification to this module.

However, in case of catastrophic failure like the crash or disconnection of an executor process, the on-disk shuffle data will no longer be available. This error will trigger a *FetchFailedException*, when the currently running tasks try to load their shuffle input. In this case, the runtime cannot simply restart only the tasks that trigger the error, it has to recompute the missing shuffle data as well. In order to do this, the current design calls for the cancellation of the erroneous stage and the restart of its parents. Because there is only two layers of dependency: the parents stages and the children stages, when we cancel the children stages, the entire cluster will be free to run the parents stages.

On the other hand, when we remove the stage barrier using the considered pipelined data-transfer policy, we can have three or more layers of dependency: the parents, the children and the grandchildren. For the purpose of clarity, we will called them $S1$, $S2$ and $S3$, respectively. $S1$, which is the parent of $S2$, has already finished successfully. $S2$ is the currently running stage, which is about to complete. $S3$ is a children of $S2$ which was started early to processed the shuffle data of $S2$. In a unmodified Spark runtime, when $S2$ encounters a a *FetchFailedException*, it will be canceled and $S1$ will be restarted as explained above. However, due to the existence of $S3$, a deadlock can happen.

When $S2$ is canceled, all its CPU slots are released so that the runtime can use them to run the tasks of $S1$. Nonetheless, the runtime is also free to used these CPU slots to execute the tasks of $S3$. The deadlock happens when all CPU slots are allocated to $S3$ instead of $S1$. Because $S2$ have not finished, $S3$'s will be stuck waiting for addition data from $S2$. However, $S2$ need the result of $S1$, which cannot be executed due to the lack of CPU slots.

This deadlock is quite likely to happen, because there is a timeout between the time when $S2$ is canceled and the time $S1$ is rescheduled. This delay is an optimization to handle multiple *FetchFailedExceptions* which arrive in quick succession with a single restart command. During this period and the time spent on the delayed scheduling algorithm, $S3$ might get to occupy all the free CPU slots and cause the starvation of $S1$.

In order to avoid this deadlock, whenever $S2$ encounters an *FetchFailedException* error and have to be canceled, we also cancel $S3$. This action free up the cluster resources to restart $S1$. This is the closest behavior to the original version of Spark. However, it also means that the CPU time spent on the $S3$ is wasted. Unlike $S2$, who gets to keep the result of the tasks which have finished before the error, $S3$ have no successful tasks and have to be restart entirely. This, however, can be avoided with a more intelligent scheduler. Because only the faulty parts of $S1$ and $S2$ need to be recomputed, it should not be necessary to fully cancel $S3$. We only need to cancel a small number of tasks in $S3$, so that we have enough CPU slots to restart $S1$ and $S2$. Unfortunately, this is difficult to implement with the current scheduler's architecture of Spark, so canceling $S3$ completely is our best choice.

IV. Evaluation & Result

A. Environment

This evaluation is performed on the INESC-ID cluster. All machines have 2 Intel Xeon 5506 CPUs with 4 cores each, and 48 GB of RAM and are connected using 1 Gbps ethernet. However, the experiments are not run directly on the the physical machines, but on OpenStack's virtual machines. The virtual machines are configured to use 4 CPU cores each and 16 GB or RAM each, which is similar to the size of an Amazon EC2 m4.xlarge VM [16] at the time of writing. All evaluation is done using 5 physical machine, each hosting only 1 VM. One virtual machine is used as the Spark's master and HDFS's namenode, while the other four are used to run Spark's workers and HDFS's datanodes. We implemented our modifications using Spark version 1.6.2. The evaluation is performed with Java 7, HDFS 2.7.2 and Ubuntu Linux 14.04.

B. Methodology

In order to evaluate the performance difference when changing Spark's execution strategy from batch to pipeline, multiple benchmarks were performed with a variety of configurations, which will be presented below. For each benchmark and configuration, we performed the following procedure:

1) Execute the benchmark multiple times on with and without our modification to Spark to support pipelining.
2) The execution time of each run is collected and grouped into two sets based on whether it makes use of our modification or not.
3) These two sets are sorted and filtered to remove the top and bottom 10 %. This step gets rid of the extreme value that could skew the result.
4) Calculate the mean, standard deviation and the 95 % confidence interval of the execution time for each set. Furthermore, we also calculated the mean difference of the two sets to quantify our improvement.
5) Perform the unequal variance t-test (Welch's t-test) [17] on the two sample sets with the null hypothesis that the two population means are equal. If this hypothesis can be rejected with 95 % confident, then we can conclude that overlapping stage execution causes a statistically meaningful change in execution time.

All benchmark applications were configured so that the number of partitions in a RDD is double the total number of CPU cores in the cluster, as recommended in Spark's developer manual.

C. Micro-benchmark: Randomize Text

We start our experimental evaluation by presenting the results achieved while using a micro-benchmark which allowed to stress, in a controlled manner, the performance of the inter-stage data-transfer mechanism. We developed a micro-benchmark to determine the effects of switching Spark's execution strategy from batch to pipeline in the ideal conditions. The requirements for this benchmark are being both computation and I/O intensive and have an adjustable number of stages. After much testing, we settled on a benchmark that works as followed:

1) Read a text file into a RDD, where each record is a line in the input file. This RDD is materialized and kept in memory to use in step 2. The reading time was recorded and subtracted from the result because it was not a part of the shuffle process and was the same whether running on our modified version of Spark or not.
2) Perform a random shuffle of the characters within each record in the RDD. This step aims to satisfy the computation-intensive condition.
3) Re-partition the resulting RDD in the previous step. Due to the random shuffle, most records will need to be transfer to a different computer, making this workload I/O intensive.
4) Repeat step 2 and 3 for as many times as necessary to achieve the required number of stages.

The experiments were performed with the input size of 0.6 GB, 1.2 GB and 1.8 GB and the number of iterations set to 4, 8 and 16.

Figure 7 and 8 describes the time during which an executor is actively running a task for the workload of 0.6 GB input and 4 iterations. The red bar indicates the entire duration of a stage, while each of the four other bars represents the duration of each executor. The gray area in a executor's bar signals that some tasks in that executor have to wait for the data produced in a preceding stage.

In figure 7, which reports the results obtained using the unmodified Spark, the executors always start together at the beginning of the stage, and a slow executor will force the others to wait for it. This limitation is removed when running the application when using the pipelined data transfer scheme: the executors can be assigned new task as soon as possible, and the stages are nicely overlapped as shown in figure 8

Figure 9 shows the difference between execution time on our platform and the original Spark, when the number of iterations is set to 16 and the input size is changed between 0.6 GB, 1.2 GB and 1.8 GB. Results indicate that around 10 to 20 % of reduction in time is achievable. However, it also shows that the performance difference drops as the

Figure 7. Randomize Text with batch execution

Figure 8. Randomize Text with pipeline execution

amount of input data increase, due to the increasing cost of I/O operation. As the amount of data needs to be shuffled increased, the cost of I/O operation overshadowed the cost of CPU operation. Because overlapping stages focuses on exploiting the free CPU time at the end of each stage, the improvement became less significant when you raise the size of data. We also witnessed similar behavior when the number of iteration is set to 4 or 8.

Figure 10 indicate a decrease in the relative execution time between running the application using our pipeline implementation and the original batch strategy, as the number of iterations increase, while the input size is fixed at 0.6 GB. These results are in line with our expectation, because the number of iterations is tied directly to the number of stages in this benchmark. In addition, the more stages we have in our application, the higher performance we can extract from the cluster by overlapping them. However, the performance did not improve linearly with the number of stages. In fact, with the same input size, doubling and quadrupling the number of iterations brought very little change in relative runtime and peaked out at 22 %.

Overall, the results obtained using our micro-benchmark show that in workloads that make intensive use of operators requiring inter-stage data transfers, a batch-based policy,

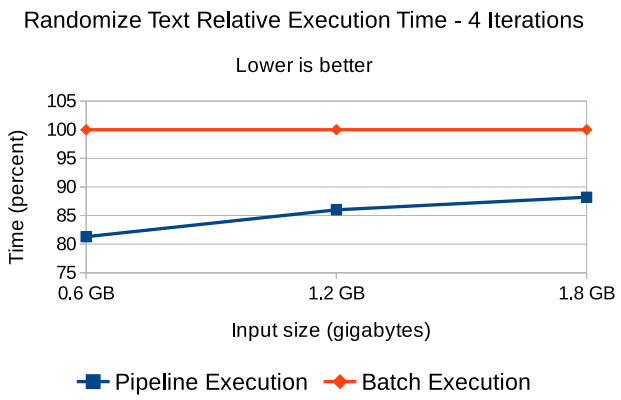

Figure 9. Comparison of Randomize Text Execution Time - 16 Iterations

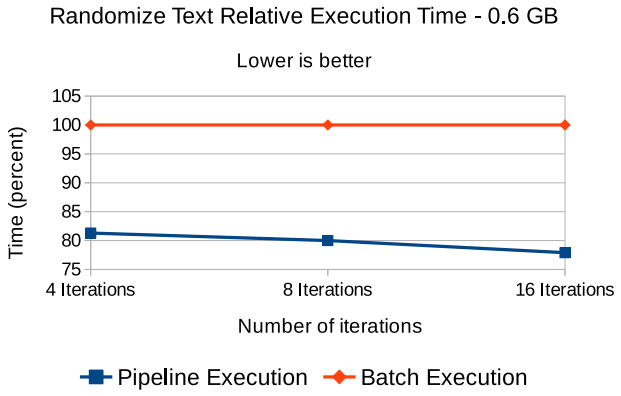

Figure 10. Relative value

such as the one currently adopted by Spark, can achieve performance that is up to 22% lower than a pipeline approach.

D. Classic two-stage benchmark: Terasort

Terasort is a benchmark to sort one Tera byte of randomly distributed data. When implemented in Spark, it consists of two stages

1) The input data is read and divided using a range partitioner. For example, in order to have n partitions, we need to choose n number $a_1, a_2, ..., a_n$ so that $a_1 < a_2 < ... < a_n$. Then the number i will belong to partition j if $a_j <= i < a_{j+1}$
2) Sort each partition and output the data

Due to limited resource, we were not able to test **Terasort** with the full 1 TB of input, which would require hundreds of machines to finish in a reasonable amount of time. Instead, we generated 3 input sets with size 1.6 GB, 3.2 GB and 6.4 GB. These sizes allowed us to repeat the application many times and collected the necessary number of samples for the statistic tests.

The result in figure 11 indicates that there are little performance different when running TeraSort using pipeline execution and batch execution. Furthermore, T-testing also confirmed that no statistical significant improvement is detected. This is probably due to the limited opportunity to overlap execution of stages, when there are only two stages involve.

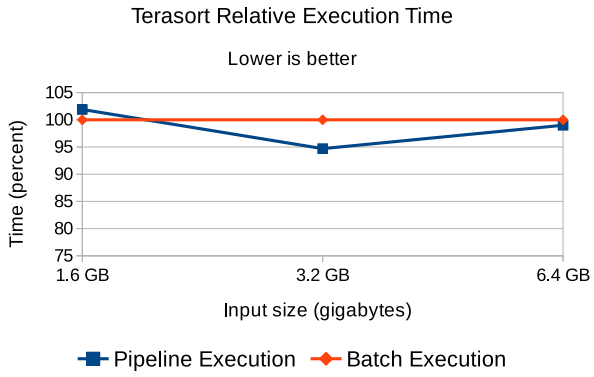

Figure 11. Comparison of Terasort Execution Time

E. Multi-stage benchmark: Pagerank

PageRank [18] was the original algorithm used by Google to rank website in their search engine results. In this algorithm, the Internet is a graph where each website is represented by a vertex and a link from website A to website B is represented by an edge from vertex A to vertex B. Each website has a ranking, calculated based upon the ranking of other websites that link to it.

In our benchmark, we use four graphs with 1, 2, 3 and 4 millions vertices, and the number of out-going edges follow the log-normal distribution. Each execution of the experiment was run using a 3-iteration version of PageRank and the total times minus the time to read the input graph was recorded.

Figure 12 shows the difference in execution time of the PageRank algorithm between the pipeline approach in our modification and the original batch approach used in Spark. We can see that the gap between our version of Spark and the original increases with the size of the graph. With the 1-million graph, the improvement is only about 5.7 %, while with the 3-million graph, this has increase to around 16.6 %. However, when the graph size reach 4 million, the improvement decrease slightly to 15.3 %.

F. Other benchmarks

We also performed addition benchmarks such as Words Count (counting words' frequency in a documents), Connected Components (find out which graph's components a vertex belongs to) and Triangles Count (find the number of triangles in a graph). When switching from batch to pipeline execution, the performance difference ranges from no change in Words Count, to a modest 5 % improvement for Connected Components, to around 20 % gain in the case of Triangles Count. Further details about these benchmarks can be found in the full text of the dissertation.

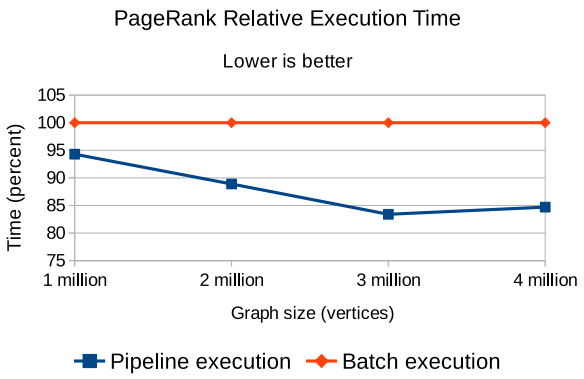

Figure 12. Comparison of PageRank Execution Time

V. Conclusions

Switching from batch to pipeline execution in Apache Spark does bring performance improvement. However, the amount of enhancement depends on the workload. When the job includes many shuffle operations, we have shown that a 10 to 20 % reduction in execution time is achievable. On the other hand, in less favorable workload such as Terasort, we might not be able to detect any difference at all.

Our modification implements a coarse-grained pipeline strategy, however a fine-grained approach will probably be even more efficient and should be further investigated. Other interesting future works include extending Spark's scheduler to take memory usage into account and adjusting this modification to support other cluster managers like YARN [19] or Meso [20].

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," vol. 51, no. 1, pp. 107–113. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[2] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *2010 IEEE International Conference on Data Mining Workshops*, pp. 170–177.

[5] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. ACM, pp. 147–156. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595641

[6] Apache Flink: Scalable Batch and Stream Data Processing. [Online]. Available: https://flink.apache.org/

[7] G. F. Pfister, "In Search of Clusters (2nd ed.)," in *In Search of Clusters (2Nd Ed.)*. Prentice-Hall, Inc., p. 36.

[8] N. P. Kronenberg, H. M. Levy, and W. D. Strecker, "VAXcluster: A Closely-coupled Distributed System," vol. 4, no. 2, pp. 130–146. [Online]. Available: http://doi.acm.org/10.1145/214419.214421

[9] C. The MPI Forum, "MPI: A Message Passing Interface," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. ACM, pp. 878–883. [Online]. Available: http://doi.acm.org/10.1145/169627.169855

[10] PVM: Parallel Virtual Machine. [Online]. Available: http://www.csm.ornl.gov/pvm/

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. ACM, pp. 59–72. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273005

[12] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale." ACM Press, pp. 423–438. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2517349.2522737

[13] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. ACM, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755940

[14] [SPARK-2387] Remove the stage barrier for better resource utilization - ASF JIRA. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-2387

[15] G. Graefe, "Volcano&#151 An Extensible and Parallel Query Evaluation System," vol. 6, no. 1, pp. 120–135. [Online]. Available: http://dx.doi.org/10.1109/69.273032

[16] EC2 Instance Types Amazon Web Services (AWS). [Online]. Available: //aws.amazon.com/ec2/instance-types/

[17] B. L. Welch, "The Generalization of 'Student's' Problem when Several Different Population Variances are Involved," vol. 34, pp. 28–35.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[19] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. ACM, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center."